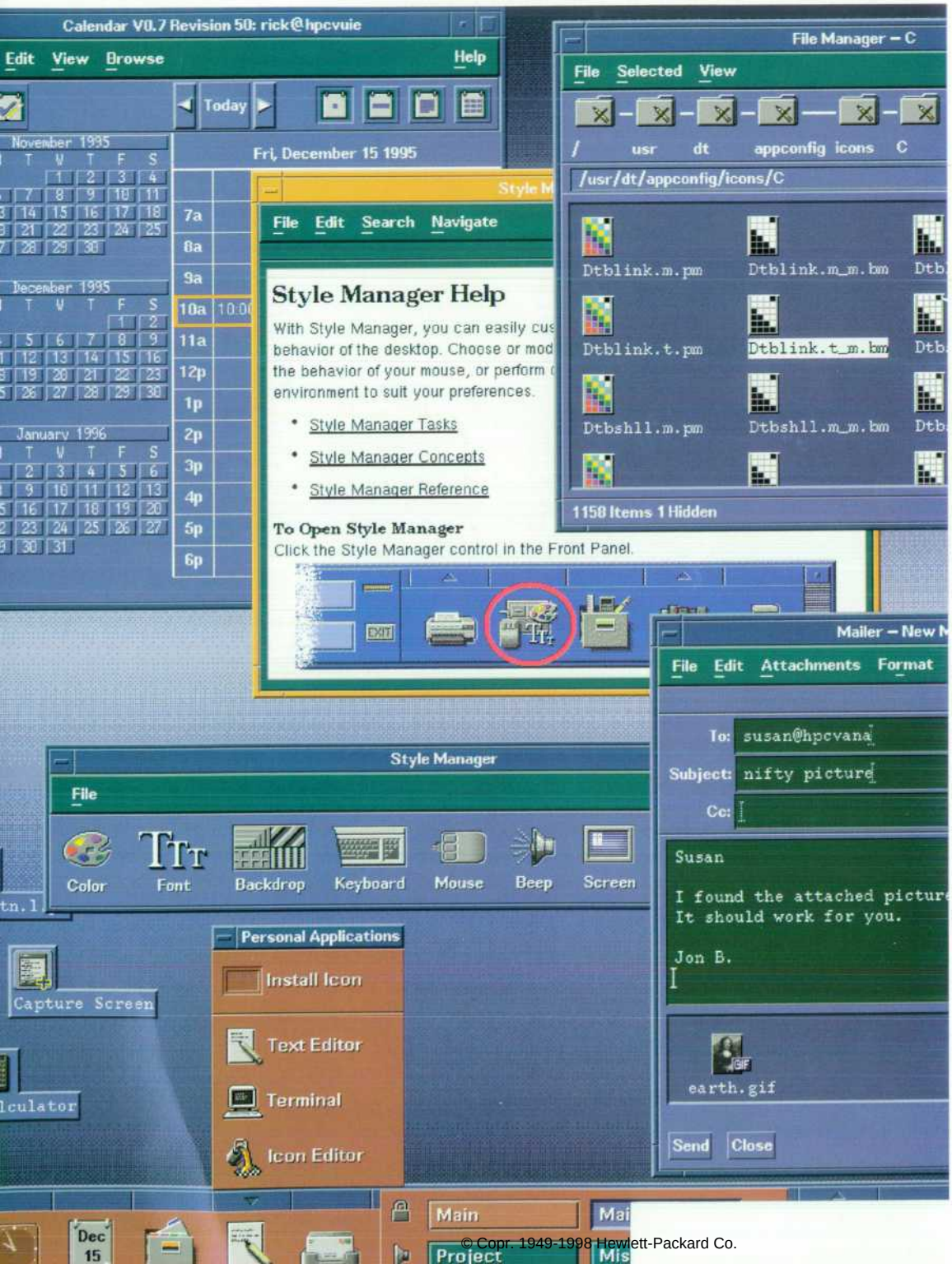
HEWLETT-PACKARD
JOURNAL
April 1996



HEWLETT PACKARD

HEWLETT-PACKARD
JOURNAL

April 1996 Volume 47 • Number 2

Articles

Departments

The Hewlett-Packard Journal is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. To receive an HP employee subscription you can send an email message indicating your HP entity and mailstop to ldc_litpro@hp-paloalto-gen13.om.hp.com. Qualified non-HP individuals, libraries, and educational institutions in the U.S. can request a subscription by either writing to: Distribution Manager, HP Journal, M/S 20BH, 3000 Hanover Street, Palo Alto, CA 94304, or sending an email message to: hp_journal@hp-paloalto-gen13.om.hp.com. When submitting an address change, please send a copy of your old label to the address on the back cover. International subscriptions can be requested by writing to the HP headquarters office in your country or to Distribution Manager, address above. Free subscriptions may not be available in all countries.

The Hewlett-Packard Journal is available online via the World-Wide Web (WWW) and can be viewed and printed with Mosaic and a postscript viewer, or downloaded to a hard drive and printed to a postscript printer. The uniform resource locator (URL) for the Hewlett-Packard Journal is http://www.hp.com/hpj/Journal.html.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presented in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, M/S 20BH, 3000 Hanover Street, Palo Alto, CA 94304 U.S.A.

In this Issue

A standard, easy-to-use, intuitive user interface for computer systems has been a goal of system developers since the days of the ENIAC* computer. The UNIX® operating system, which began life as a system for engineers only, has been a consistent target for various user-interface schemes.

Before graphical user interfaces (GUIs) became commonplace, users interacted with computer systems via a character mode command-line interface. For UNIX systems, this command-line interface mode continued until the 1980s with the arrival of proprietary window systems. By 1988 the X Window System had been adopted as the standard window system for UNIX systems. Toolkits were created that allowed UNIX vendors to create as many different proprietary user interface environments as there were vendors. The article on page 6 introduces eight articles that describe how four UNIX vendors, who normally compete in the marketplace, collaborated on a user interface environment for UNIX platforms called the Common Desktop Environment, or CDE. This article explains how this environment is seen from three different viewpoints: developers who write applications to run in CDE, system administrators who must install and maintain these applications, and finally, end users who use these applications.

Since UNIX systems are highly networked, it is desirable that a desktop environment allow network transparency—the ability to launch applications and access data without worrying about where in the network these items are located. Thus, when the user selects an application (by double-clicking an icon) that happens to be on a remote system, the user environment automatically establishes links to the remote application server, allowing the user to run the application as if it were located on the local workstation. The article on page 15 describes the underlying mechanisms that link icons to applications, and the tools that enable system administrators to integrate applications into the desktop environment.

In most cases today the icons on a graphical desktop are fairly intuitive. For example, if you drop a document on a printer icon very likely the document will be sent to a printer of your choice. The article on page 24 describes the APIs (application programming interfaces) and databases responsible for defining the look and behavior of icons in the Common Desktop Environment.

The world of online help has evolved from simple out-of-context cryptic messages to media-rich, context-sensitive help messages. As the article on page 38 explains, the CDE help system is based on the easy-to-use HP VUE 3.0 help system. Like HP VUE 3.0, the CDE help system provides a language (HelpTag) for authors to develop help messages and tools for programmers to integrate customized help messages into their applications. The main difference between CDE help and HP VUE 3.0 help is the delivery format for help files. CDE help uses the semantic delivery language (SDL) as a delivery format for help files. SDL focuses on the semantics of a document.

Many users are content with the special menu and icon customizations they have in their current HP VUE interface. Therefore, to allow users to keep their menu and icon customizations in CDE, a collection of utilities are provided to translate HP VUE customizations into CDE equivalents. These utilities are described on page 29.

As mentioned above, the CDE project was a joint engineering effort between four companies that typically compete in the marketplace. The companies are HP, IBM, Sun Microsystems, and Novell. All four of these companies produce computer systems that use the UNIX operating system as their system platform. Because of different cultures and development strategies, the joint effort presented some interesting and unique challenges. In the article on page 50, the author describes the mechanisms and procedures that had to be put in place to manage the CDE project. Because of the different development strategies, test tools, and verification methods, the test team had the greatest challenge in this multicompany project. As the article on page 54 states, to ensure quality in the final product, strict guidelines were established at the beginning of the project. For example, rules were established for test coverage and assigning reponsibility when defects were found.

* The ENIAC (Electronic Numerical Integrator And Calculator) was developed in 1946, and is considered to have been the first truly electronic computer.

One of the tools that made testing the graphical user interface of CDE much easier is a test tool called Synlib. Typically, an image capture and compare technique is used to verifiy the various windows (screen states) of a GUI. However, this technique is sometimes prone to inaccuracies. Although the image capture technique was used for CDE testing, the majority of GUI testing used Synlib. Synlib reduces the need to use image capture for checking screen states because it employs a combination of position independent and position dependent data to manipulate and verify desktop items. Synlib is introduced in the article on page 54 and described in the article on page 62.

For a mobile telephone to be competitive today it must supply full output power at a supply voltage of 5.4 volts (five NiCad cells) with 40% efficiency and 0-dBm input power. It should also be inexpensive, small, have a long talk time, and be able to be manufactured in volume. These characteristics determine the specifications for the power module in a mobile telephone. The power module in a mobile telephone is the output stage of the RF (radio frequency) amplification chain of the telephone. The article on page 66 describes the design of a 3.5-watt power module for a GSM (Global System for Mobile Communications) mobile telephone, which satisfies all the specifications mentioned above.

The article on page 73 is a good example of the wide range of products offered by HP. The HP G1009A C-terminal protein sequencing system is a complete, automated system for the carboxy-terminal amino acid sequence analysis of protein samples. Using adsorptive sample loading and a patented sequencing chemistry, the HP G1009A is capable of detecting and sequencing through any of the 20 common amino acids.

Time-domain reflectometry (TDR) is commonly used for determining the characteristic impedance of a transmission line or quantifying reflections caused by discontinuities along or at the termination of a transmission line. However, as the article on page 83 explains, TDR can also be used to measure the capacitance or inductance of devices or structures while they reside in the circuit. The typical inductance-capacitance-resistance (LCR) meter cannot make these measurements accurately. The article shows how the HP 54750A oscilloscope and the HP 54754A TDR plug-in card can be used to make these measurements.

C.L. Leath
Managing Editor

Cover

A screen showing a typical collection of icons, panels, windows, and dialog boxes that make up the graphical user interface of the Common Desktop Environment.

What's Ahead

In the June issue we'll have four articles on the HP 16505A prototype analyzer and articles on the HP PalmVue mobile clinical patient information transmission system, the HP Omnibook Pentium™ PCI-based notebook computer, and the HP 38G graphing calculator for math and science classes. There will also be a paper on developing an application server in a highly networked environment.

A Common Desktop Environment for Platforms Based on the UNIX® Operating System

User interface technologies from four companies have been combined to create a single UNIX desktop standard that provides a common look and feel for end users and a common set of tools for system administrators and application developers.

by Brian E. Cripe, Jon A. Brewster, and Dana E. Laursen

Until the early 1980s most users interacted with their computers via character-mode interfaces—they typed in commands. What happened to change all this was the arrival of proprietary window systems. HP's first window system was embedded in the Integral Personal Computer.[1] By 1988 the X Window System had been adopted as a standard for machines running the UNIX operating system. However, available user interface toolkits, such as HP's CXI widgets, were proprietary. These toolkits provided items such as scroll bars and pop-up menus, and they allowed software developers to create applications that had a consistent look and feel. By 1990 two stable toolkits had emerged, OSF/Motif and OpenLook.†

The stage was now set for proprietary user environments. A user environment is a collection of programs used by the end user to manage files, invoke applications, and perform routine tasks such as edit text files and send and receive email. HP delivered its first version of HP VUE (Visual User Environment)[2] in 1990 with subsequent upgrades continuing to this day.

In March of 1993 representatives from Hewlett-Packard, IBM, Sun Microsystems, and Novell agreed to create a common user environment for UNIX platforms (see the article on page 50). This joint initiative resulted in the specification and development of the Common Desktop Environment (CDE). CDE accomplishes two things: first, it adopts OSF/Motif as the principal user interface toolkit for UNIX systems, and second, it establishes this rich new environment and framework as a standard user environment.

CDE is based on the X Window System from the X Consortium and the Motif graphical user interface from the Open Software Foundation. Fig. 1 shows how these technologies fit together.

The X Window System (X) components include:

- X server. This program writes directly to the user's display hardware.
- Xlib. This is a library of function calls for communicating with the X server. Xlib deals with low-level concepts such as rectangles, arcs, and fonts. It does not know about higher-level concepts such as menus and scroll bars (i.e., interface widgets).
- X protocol. This is the data stream that communicates between Xlib and the X server. This data stream can be passed across a network, which gives X the ability to run an application on one system and display the user interface to another system.
- Xt. This is the X toolkit, which provides a framework for defining and integrating user interface widgets.

The Motif component is Xm, which is the Motif library that provides a rich collection of user interface widgets such as dialog boxes, menus, buttons, scroll bars, and text editing panes.

Different Views of CDE

The rest of this article provides an overview of CDE from three different perspectives: the end user who uses CDE but does not care about its internal design, the software developer who is writing applications that need to be integrated with CDE, and the system administrator who is responsible for managing systems that run CDE.

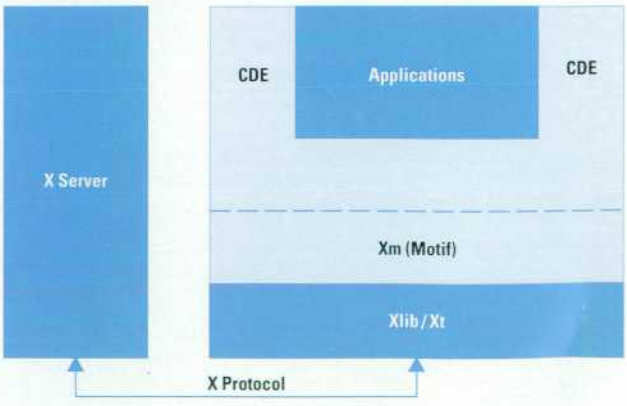


Fig. 1. CDE architecture.

† OpenLook is the X Window System toolkit from Sun Microsystems.

End-User's View

Putting together a user environment and application framework such as CDE always forces one to be precise about requirements. Many of the driving inputs for the CDE project turned out to be based on the following end-user requirements:

- Completeness. CDE needs to be a full-service environment covering everything from login to logout. This includes security and authentication, windowing, application launching, file management, email, and so on.
- Integration. CDE parts and services need to work together seamlessly. For example, it should be possible to mail a meeting notice with a calendar appointment in it, allowing the recepients to add the event easily to their calendars. Also, CDE utilities need to use CDE APIs (e.g., help, drag and drop, etc.) not only to be consistent with each other, but to be showcase components showing the results of proper use.
- Compatibility. Previous applications need to continue to run. This is true for OpenLook, Motif, and terminal-based software. For HP VUE users, we were very interested in ensuring that conversion tools could be created to move configuration information into CDE.
- Ease of use. The resulting environment needs to be guided by a standard set of usability principles and tested for usability defects. This work took place at the early stages and during every step of the CDE project.

Getting agreement on these fundamental end-user requirements was critical given the nature of our extended multi-company development team. Many of the more difficult project decisions required coming back to these basics. For example, the drag and drop architecture had to be reworked several times to accomplish the integration ambitions of the team.

The cover of this issue and Fig. 2 show a typical CDE user interface, and Table I lists all the end-user components, some of which are shown in Fig. 1.

Basic End-User Tasks

The first thing a user does when approaching CDE is log in. It is the CDE login service that authenticates and authorizes user access to the system. This gatekeeper for the UNIX system then invokes the basic session components such as the window manager and file manager. The user's previous session can also be automatically restored. This allows items such as running applications, color settings, and the graphical desktop arrangement to be retained from the logout of the previous session.

Once the user is logged in, the file manager is used to find and organize data files. Files can be dragged out of the file manager and dropped in many interesting places such as the printer. Files can be moved and copied by dragging them about. File icons can be placed on the main screen as if the screen were a desktop. A file's type (document, spreadsheet, image, etc.) can be determined by its icon, and various type-specific actions can be invoked via a pop-up menu that is made available for each icon. For example, editing a graphics image and viewing that same image might require two different applications and thus two different actions in the pop-up actions menu. The file manager also allows different views of the file directories, such as a tree view (containment hierarchy) and a full file properties view (size, security, etc.). Files can be found by manually browsing through the

Fig. 2. A typical CDE user interface.

Table I
End-User Components

Component	Function
Login Manager	Graphical login and authentication
Session Manager	Logout to login session save and restore
Window Manager	X11 Windows compliant window manager
Workspace Manager	Window grouping and task switching
Front Panel	Ubiquitous services access and workspace switching
Style Manager	Customization and configuration
File Manager	File manipulation services
Application Manager	Application discovery and launch services
Text Editor	ASCII text file editor
Icon Editor	File type icon creator and editor
Mailer	Full MIME compliant multipart email facility
Calendar	Personal and group time manager
Calculator	Financial, logical, and scientific modes
Terminal Emulator	VT220 type character mode support
Action Creator	Linking behavior to specific icons
Print Manager	Multiprinter-aware print submission tool

directories or via a finding service that works with name or content matching. A found file can then be automatically placed on the desktop.

Applications can be launched in a number of ways. Simply double-clicking a data file icon and allowing CDE to launch the correct application with that data file as its argument is usually the most convenient approach. However, sometimes running an application directly is more appropriate. The applications themselves also show up as icons that can be moved about, organized, and double-clicked for invocation. These application icons are initially found in the application manager, which is really just a file manager window onto the applications directory. A user can drag the application icons into any other directory or even onto the desktop. For important and commonly used applications, the icon can be directly installed into the front panel for easy access.

The article on page 15 describes the data structures involved in linking applications, icons, and actions together.

Task management in CDE is much like any other windowing environment in that it has various ways to control the windowing behavior of the running applications. However, there is a major enhancement in CDE called *workspaces*. The multiple-workspace paradigm supported by CDE allows a user to switch between multiple screens full of application windows. This allows each collection of windows to be treated as a group (the default is four workspaces). End users can use workspaces to collect together groups of windows for specific tasks. For example, one workspace could contain all the windows associated with a remote system, while another workspace could contain windows for the user's desktop publishing tools.

Workspace switching is very fast, typically less than a few tenths of a second, and is accomplished by pressing a single button within the front panel. Each workspace has its own backdrop, which is usually color-coded to match its button. The workspaces are also named and can be created and destroyed at any time by the user.

The front panel is usually located at the bottom of the screen (see Fig. 2). It is available in all workspaces and provides access to important and frequently used services. These services include:

- Clock and calendar functions
- Personal file space (file manager access)
- General application space (application manager access)
- High-use application and data file access
- Mail facilities
- Workspace management controls
- Printer access
- CDE configuration controls
- Help facilities
- Trash can for deleting files
- Screen locking and logout services.

The front panel is a unifying tool for collecting important services that are otherwise left scattered and hard to find. It is also a distinctive visual feature known to the development team as a *signature visual* in that it distinguishes the machine running CDE from other machines.

End-User CDE Utilities

CDE has a number of utility programs that support specific end-user tasks. The following are some of these utilities.

Text Editor. This is a simple ASCII text editor that is neatly integrated into CDE so that drag and drop, help, and general Motif text behavior are seamlessly available to the end user. Text formatting, spell checking, and cut and paste services are also available.

Mailer. This is a highly integrated set of services that has full drag and drop behavior with respect to folders, messages, and attachments. The following scenario illustrates the services integrated with the mailer.

The front panel's mail icon is pressed, bringing up the user's in-box. A message is dragged from the in-box to the front panel's printer icon. Another message is dragged to a mail folder icon within a file manager for saving. A file is dragged from the file manager to the mail icon on the front panel for sending. Finally, a message being read has a graphics attachment that is double-clicked to invoke the graphics viewer.

Thus, from the CDE mailer the user can print a message, save a message to a file, view the graphics in a message, and compose and send a message.

Calendar. The calendar facility is a personal tool for managing time and to-do items. It is "group aware" so that the user can examine and search for meeting opportunities with colleagues. Individuals can control the privacy of their own schedules. Meetings can be emailed via the drag and drop

mechanism, and the calendar view can be flipped between day, week, month, and six-month views.

Terminal. This tool supports character-mode applications (some of which predate window systems). The CDE terminal emulator behaves like a DEC VT220 terminal with minor changes consistent with ANSI and ISO standards. Full cut and paste behavior with the rest of the desktop is built in. The core feature of this emulator traces its ancestry back to HP's Integral Personal Computer, which had HP's first windowing system and thus HP's first terminal emulator for the UNIX operating system.

Software Developer's View of CDE

To the software developer, CDE is composed of two distinct components: the X/Open® standard and CDE product implementations. The X/Open standard defines the components that must be present on any system that claims to be CDE-compliant. HP's CDE product, like CDE products from other vendors, must support the interfaces defined by the X/Open CDE standard, but may contain additional functionality. For example, many vendors have enhanced CDE to provide backward compatibility with previous proprietary products. Software developers should be cautious when using features of a CDE product that are not part of the X/Open standard because they may not be portable to all CDE systems.

The major benefits that CDE provides to the developer are:

- A single GUI toolkit (OSF/Motif) that is supported by all major UNIX vendors
- Tools and libraries to help build an application
- Mechanisms to integrate an application with the desktop environment
- Mechanisms to integrate applications with each other.

Table II lists the components available in CDE that enable developers to integrate their applications into CDE. Appendix A, on page 11, contains a complete list of all the CDE APIs, which enable developers to build applications based on CDE.

CDE defines three levels of application integration from which the developer can choose: basic, recommended, and optional. Basic integration consists of the minimal integration steps that allow a user to access an application from the desktop environment instead of a command prompt in a terminal window. Recommended integration requires more effort by the developer but allows the application to be fully consistent with CDE and other CDE applications. The final level, optional integration, is not necessary for most applications but is useful for applications that need to perform specialized tasks.

Basic Integration

Basic integration allows an application and its data files to be managed by CDE. This management includes:

- Finding the application and invoking it using an icon in the application manager
- Identifying the application's data files with a unique icon
- Loading a data file into the application by dragging and dropping the file icon on the application icon
- Invoking the application to print a data file by dragging and dropping the file icon on a printer icon
- Using the style manager to specify colors and fonts for the application

Table II
Developer Components

Component	Purpose
Help System	Viewer, API, and administration tools
Motif Toolkit	OSF/Motif version 1.2.3 plus some 1.2.4 repairs
Custom Widgets	SpinButton and ComboBox (taken from Motif 2.0)
Terminal Widget	For adding terminal emulation to an application
Dtksh	A GUI dialoging and scripting facility
Data Interchange	Drag and drop conventions, protocols, and APIs
ToolTalk®	A standard general-purpose message passing service that enables tight integration between separate applications and CDE components
Data Typing	API for access to the desktop engine typing services
Actions	API for access to the desktop invocation services
Font Guidelines	Conventions for standard font interactions
Internationalization Guidelines	Overview and reconciliation of relevant standards and style guides
Client/Server Guidelines	Network execution and distribution model

- Locating information about the application in the help manager.

Basic integration can be accomplished without any modifications to the application's source or executable files. The steps for performing basic integration include:

- Defining an application group for the application manager that will hold the application
- Defining the application's icons and giving them the correct double click and drag and drop behavior by creating new CDE actions and data types
- Removing color and font specifications from the application's defaults file so that it will use the default colors and fonts specified by the CDE style manager
- Installing configuration files for the application in the standard file system locations and registering it with the desktop using the dtappintegrate command (This command is typically invoked by the application's installation script.)
- Creating and installing any appropriate application information as desktop help files.

Recommended Integration

Recommended integration includes additional steps that are necessary to make the application fully integrated into CDE and consistent with other applications. This level of integration requires modifications to the application's source code, but in return it provides the following benefits:

- The user can access context-sensitive online help from within the application using the CDE help system. To achieve this the application must use the help system API.

- The application can achieve tight integration with other applications using ToolTalk messages. For example, an editor application that supports the media exchange message suite can be used by other applications for editing data objects. To achieve this the application must use the ToolTalk messaging API and support one or more of the standard message suites.
- The user can log out with the application running and upon login the application will be automatically restarted and restored to its previous state. To achieve this the application must be able to read and write session files that define the application's state, and it must use the session management API for controlling this behavior.
- The user can move data into or out of a running application using the drag and drop facility. To achieve this the application must use the drag and drop API.
- The application user interface can be translated into other languages without modifying the source code for the application. To achieve this the application must follow the internationalization guidelines.
- The application can be displayed on a remote workstation or an X terminal and be assured of getting the expected fonts. To achieve this the application must access its fonts using the standard font names defined in the font guidelines.

Optional Integration

Applications with unique needs may choose to use the optional CDE integration facilities. Applications are not required or expected to use any of these facilities, but some applications will find them useful. These facilities include:

- Additional Motif widgets such as SpinBox and ComboBox
- Data typing functions that enable an application to determine the type and attributes of files and other data items in a manner consistent with CDE and other applications
- Action invocation functions that enable an application to invoke other applications
- Monitor and control functions for the placement of applications in the user's workspaces
- A terminal emulator widget that can be used to add a conventional UNIX command window to the application
- A text editor widget that allows adding a text editing window to the application, which is more powerful than the standard Motif text editor widget
- An API to access calendar and scheduling capabilities, which is an implementation of the X.400 association calendaring and scheduling API 1.0
- An enhanced version of Korn shell which provides access to CDE APIs from an interpreted script language.

More information about these integration techniques can be found in references 3 and 4.

System Administrator's View of CDE

CDE greatly simplifies the burden of a UNIX system administrator because it provides a consistent set of capabilities and configuration mechanisms across virtually all UNIX systems. Tasks that an administrator of a CDE system might perform include configuring the behavior of CDE, administering CDE in a networked environment, and administering applications.

Configuring CDE. CDE is a highly configurable environment. Many of the customizations that a user can choose to do to configure a personal environment can also be done by a system administrator for all users. Some examples of possible configuration changes include the ability to:

- Customize the appearance of the login screen
- Modify the set of applications that get automatically started when a user first logs in
- Add or remove printer icons from the print manager
- Customize the contents of the front panel
- Lock all or portions of the front panel so that they cannot be modified by the user
- Customize the set of controls embedded in window frames and modify their behavior
- Modify the menus available from the root window of the display
- Modify the keyboard bindings and accelerator keys used by applications
- Customize the default fonts, colors, and backdrops used by CDE and applications.

For more information on any of these tasks see reference 4.

Administering CDE in a Networked Environment. CDE is designed to work well in a highly networked environment. The architecture of the desktop lets system administrators distribute computing resources throughout the network, including applications, data files for applications, desktop session services (desktop applications such as the login manager and file manager), and help services. Help data files can be put on a central help server.

Typical tasks performed by the administrator of a network running CDE include:

- Installing the operating system and CDE on a network of systems, some of which might perform specialized tasks such as act as an application server
- Configuring the login manager so that workstations or X terminals have login access to the appropriate set of systems
- Configuring the distributed file system so that all systems have access to the necessary set of data files
- Installing and configuring devices such as printers so that they are accessible from the desktop environment
- Configuring application servers that run applications on behalf of other systems in the network
- Configuring other servers such as database servers or help servers.

CDE includes a number of daemons. System administrators often do not need to be aware of these daemons because they are installed and configured automatically when CDE is installed. However, in some situations system administrators may need to use the information in the manuals and man pages to create some nontypical configurations.

These daemons include:

- dtlogin. The login manager, which provides login services to the workstation or X terminal
- dtspcd. The subprocess control daemon, which provides remote command invocation
- rpc.ttdbserver. The ToolTalk database server, which is used by the ToolTalk messaging system and performs filename mapping
- ttsession. The ToolTalk message server, which provides message passing

- rpc.cmsd. The calendar daemon, which manages the calendar databases.

More information about these daemons can be found in reference 3.

Administering Applications. The networking capabilities of the HP-UX* operating system, the X Window System, and CDE can be used to create many different application execution configurations. The simplest configuration is local application execution in which applications are installed on the local disk of a workstation and executed locally.

A variation of this configuration is to install applications on a disk on a central file server and then mount that disk on other workstations. Each workstation accesses the application's executable and configuration files across the network, but executes them locally. This configuration reduces the total amount of required disk space because multiple workstations are sharing a single copy of the application files.

Another approach is to use centralized application servers. This configuration uses the client/server capabilities of the X Window System to execute the application on one system and display its user interface on another workstation or X terminal.

Application servers are a good solution to the problem of giving users access to applications that have special run-time requirements. For example, if users need access to an application that only runs on HP-UX 8.0, an HP-UX 8.0 application server can be created and accessed from workstations running HP-UX 9.0.

CDE makes these distributed application environments simple to use by representing all applications as icons in the application manager. The user does not need to know or care whether the application is installed locally or on a remote application server.

CDE also makes these distributed configurations easy to create and administer. Applications are installed the same way whether they will be used locally or accessed remotely. When a workstation is configured to access another system as an application server, all of the applications on that system that have been registered with CDE automatically become available. The article on page 15 provides a more detailed discussion about CDE application administration tools.

Summary

The HP VUE user will find much to appreciate in CDE. CDE retains the best end-user features of HP VUE, such as workspaces and the iconic desktop behavior. CDE adds many new end-user services, such as an integrated mailer and a calendar system. The system administrator gets a rich and new standard set of configuration options that also shares much of the HP VUE approach. A software developer has optional access to a new programming framework to take advantage of deep environment integration. Other than the help facility, these programming services were not available as part of HP VUE.

References

1. *Hewlett-Packard Journal*, Vol. 36, no. 10, October 1985, pp. 4-35.
2. C. Fernandez, "A Graphical User Interface for a Multimedia Environment," *Hewlett-Packard Journal*, Vol. 45, no. 2, April 1994, pp. 20-22.
3. *CDE Programmer's Overview*, Hewlett-Packard, Part Number B1171-90105, January 1996.
4. *CDE Advanced User's and System Administrator's Guide*, Hewlett-Packard, Part Number B1171-90102, January 1996.

Appendix A: CDE Application Programming Interfaces

This appendix lists the CDE libraries and header files that contain the CDE APIs.

Desktop Services Library (libDtSvc)

Desktop Initialization APIs. The desktop services library must be initialized with DtAppInitialize() or DtInitialize() before an application can use the APIs for action invocation, data typing, drag and drop, screen saving, session management, or workspace management.

- #include <Dt/Dt.h>
- Dt(5): Miscellaneous desktop definitions.
- DtAppInitialize(3), DtInitialize(3): Desktop services library initialization functions.

Action Invocation APIs. These APIs provide applications access to the desktop action database to query action attributes and to invoke actions. The DtActionInvoke() function selects the correct desktop action to invoke based on its arguments and actions in the database. The DtDbLoad() and DtDbReloadNotify() functions apply to the shared database for actions and data types.

- #include <Dt/Action.h>
- DtAction(5): Action service definitions
- DtActionCallbackProc(3): Notifies application that the status of an action has changed
- DtActionDescription(3): Obtains the descriptive text for a given action
- DtActionExists(3): Determines if a string corresponds to an action name
- DtActionIcon(3): Gets the icon information for an action
- DtActionInvoke(3): Invokes a CDE action
- DtActionLabel(3): Gets the localizable label string for an action
- DtDbLoad(3): Loads the actions and data types database
- DtDbReloadNotify(3): Registers callbacks for changes to actions and data types database.

Data Typing APIs. Data typing APIs provide applications with access to the desktop data type database to query data type attributes and to determine the data type of files, buffers, and data.

- #include <Dt/Dts.h>
- DtDts(5): Provides data typing definitions

- DtDtsBufferToAttributeList(3): Gets a list of data attributes for a byte stream
- DtDtsBufferToAttributeValue(3): Gets a single data attribute value for a byte stream
- DtDtsBufferToDataType(3): Gets the data type for a byte stream
- DtDtsDataToDataType(3): Gets the data type for a set of data
- DtDtsDataTypeIsAction(3): Determines if the data type is an action
- DtDtsDataTypeNames(3): Gets a list of available data types
- DtDtsDataTypeToAttributeList(3): Gets a list of attributes for a data type
- DtDtsDataTypeToAttributeValue(3): Gets an attribute value for a specified data type
- DtDtsFileToAttributeList(3): Gets a list of attributes for a file
- DtDtsFileToAttributeValue(3): Gets a specified attribute value for a file
- DtDtsFileToDataType(3): Gets a data type for a file
- DtDtsFindAttribute(3): Gets a specified list of data types
- DtDtsFreeAttributeList(3): Frees a list of data attributes
- DtDtsFreeAttributeValue(3): Frees a data attribute value
- DtDtsFreeDataType(3): Frees a data type pointer to memory
- DtDtsFreeDataTypeNames(3): Frees a list of data type names
- DtDtsIsTrue(3): Returns a Boolean value associated with a string
- DtDtsLoadDataTypes(3): Loads and initializes the data types database
- DtDtsRelease(3): Frees memory associated with the data types database
- DtDtsSetDataType(3): Sets the data type of a directory.

Drag and Drop APIs. The drag and drop APIs are a convenience and policy layer on top of Motif 1.2 drag and drop. The drag and drop APIs manage the configuration and appearance of drag icons, define a transfer protocol for buffers, enable animation upon drop, provide enumeration of targets for text and file transfers, allow dual registration of text widgets for text and other data, and provide prioritized drop formats.

- #include <Dt/Dnd.h>
- DtDnd(5): Provides drag and drop definitions
- DtDndCreateSourceIcon(3): Creates a drag source icon
- DtDndDragStart(3): Initiates a drag
- DtDndDropRegister(3): Specifies a drop site
- DtDndDropUnregister(3): Deactivates a drop site.

Screen Saver APIs.

- #include <Dt/Saver.h>
- DtSaver(5): Provides screen saver definitions
- DtSaverGetWindows(3): Gets the list of windows for drawing by a screen saver application.

Session Management APIs.

- #include <Dt/Session.h>
- DtSession(5): Provides session management services definitions
- DtSessionRestorePath(3): Gets a path name for the application's state information file
- DtSessionSavePath(3): Gets a path name for saving application state information.

Workspace Management APIs. The workspace management APIs provide functions to access and modify workspace attributes and to request notification of workspace changes.

- #include <Dt/Wsm.h>
- DtWsm(5): Workspace manager definitions
- DtWsmAddCurrentWorkspaceCallback(3): Adds a callback to be called when the current workspace changes
- DtWsmAddWorkspaceFunctions(3): Adds workspace functions for a window
- DtWsmAddWorkspaceModifiedCallback(3): Adds a callback to be called when any workspace is changed
- DtWsmFreeWorkspaceInfo(3): Frees workspace information
- DtWsmGetCurrentBackdropWindow(3): Gets the backdrop window for the current workspace
- DtWsmGetCurrentWorkspace(3): Gets the current workspace
- DtWsmGetWorkspaceInfo(3): Gets detailed workspace information
- DtWsmGetWorkspaceList(3): Gets the names of the currently defined workspaces
- DtWsmGetWorkspacesOccupied(3): Gets the workspaces in which a window resides
- DtWsmOccupyAllWorkspaces(3): Puts a window into all workspaces
- DtWsmRemoveWorkspaceCallback(3): Removes a workspace callback
- DtWsmRemoveWorkspaceFunctions(3): Removes a window's workspace function
- DtWsmSetCurrentWorkspace(3): Sets the current workspace
- DtWsmSetWorkspacesOccupied(3): Sets the workspaces in which a window resides.

Help Widget Library (libDtHelp)

Help Utility APIs. These APIs are used to manage application help.

- #include <Dt/Help.h>
- DtHelp(5): Help services definitions
- DtHelpReturnSelectedWidgetId(3): Selects a widget or gadget
- DtHelpSetCatalogName(3): Assigns the name of the message catalog to use for help services

HelpDialog Widget API. The DtHelpDialog widget provides users with the functionality for viewing and navigating structured online information (CDE help volumes). This functionality includes text and graphics rendering, embedded hypertext links, and various navigation methods to move through online help information. The widget supports rendering of CDE help volumes, system manual pages, text files, and character string values.

- #include <Dt/HelpDialog.h>
- DtHelpDialog(5): DtHelpDialog definitions
- DtCreateHelpDialog(3): Creates a general DtHelpDialog widget
- DtHelpDialog(3): The DtHelpDialog widget class.

HelpQuickDialog Widget APIs. The DtHelpQuickDialog widget provides users with the same functionality as the DtHelpDialog widget. The difference here is that the functionality is for the quick dialog widget.

- #include <Dt/HelpQuickD.h>
- DtHelpQuickD(5): DtHelpQuickDialog definitions
- DtCreateHelpQuickDialog(3): Creates a DtHelpQuickDialog widget
- DtHelpQuickDialog(3): The DtHelpQuickDialog widget class
- DtHelpQuickDialogGetChild(3): Gets the child of a DtHelpQuickDialog widget.

Terminal Widget Library (libDtTerm)

Terminal Widget APIs. The DtTerm widget provides the core set of functionality needed to emulate an ANSI X3.64-1979- and ISO 6429:1992(E)-style terminal, such as the DEC VT220. This functionality includes text rendering, scrolling, margin and tab support, escape sequence parsing, and the low-level, operating-system-specific interface required to allocate and configure a pty or streams pseudoterminal device and write to the system's utmp device.

- #include <Dt/Term.h>
- DtTerm(5): DtTerm widget definitions
- DtCreateTerm(3): Creates a DtTerm widget

- DtTerm(3): DtTerm widget class
- DtTermDisplaySend(3): Sends data to a DtTerm widget's display
- DtTermInitialize(3): Prevents accelerators from being installed on a DtTerm widget
- DtTermSubprocReap(3): Allows a DtTerm widget to clean up after subprocess termination
- DtTermSubprocSend(3): Sends data to a DtTerm widget's subprocess.

Desktop Widget Library (libDtWidget)

Editor Widget APIs. The DtEditor widget supports creating and editing text files. It gives applications running in the desktop environment a consistent method for editing text data. The widget consists of a scrolled edit window for text, dialogs for finding and changing text, and an optional status line. Editing operations include finding and changing text, simple formatting, spell checking, and undoing the previous editing operation.

- #include <Dt/Editor.h>
- DtEditor(5): Editor widget definitions
- DtCreateEditor(3): Creates a DtEditor widget
- DtEditor(3): DtEditor widget class
- DtEditorAppend(3): Appends data to a DtEditor widget
- DtEditorAppendFromFile(3): Appends data from a file into a DtEditor widget
- DtEditorChange(3): Changes one or all occurrences of a string in a DtEditor widget
- DtEditorCheckForUnsavedChanges(3): Reports whether text has been edited
- DtEditorClearSelection(3): Clears the primary selection in a DtEditor widget
- DtEditorCopyToClipboard(3): Copies the primary selection in a DtEditor widget to the clipboard
- DtEditorCutToClipboard(3): Copies the primary selection in a DtEditor widget to the clipboard and deletes the selected text
- DtEditorDeleteSelection(3): Deletes the primary selection in the DtEditor widget
- DtEditorDeselect(3): Deselects the current selection in a DtEditor widget
- DtEditorDisableRedisplay(3): Temporarily prevents visual update of a DtEditor widget
- DtEditorEnableRedisplay(3): Forces the visual update of a DtEditor widget
- DtEditorFind(3): Searches for the next occurrence of a string in a DtEditor widget
- DtEditorFormat(3): Formats all or part of the contents of a DtEditor widget
- DtEditorGetContents(3): Retrieves the contents of a DtEditor widget
- DtEditorGetInsertionPosition(3): Retrieves the position of the insert cursor in a DtEditor widget
- DtEditorGetLastPosition(3): Retrieves the position of the last character in a DtEditor widget
- DtEditorGetMessageTextFieldID(3): Retrieves the widget ID of the message text field in the DtEditor status line
- DtEditorGetSizeHints(3): Retrieves sizing information from a DtEditor widget
- DtEditorGoToLine(3): Moves the insert cursor for a DtEditor widget to a specified line
- DtEditorInsert(3): Inserts data into a DtEditor widget
- DtEditorInsertFromFile(3): Inserts data from a file into a DtEditor widget
- DtEditorInvokeFindChangeDialog(3): Displays the DtEditor widget dialog for searching and replacing text
- DtEditorInvokeFormatDialog(3): Displays the DtEditor widget dialog for choosing formatting options
- DtEditorInvokeSpellDialog(3): Displays the DtEditor widget dialog for checking text for spelling errors
- DtEditorPasteFromClipboard(3): Inserts the clipboard selection into a DtEditor widget
- DtEditorReplace(3): Replaces a portion of the contents of a DtEditor widget
- DtEditorReplaceFromFile(3): Replaces a portion of the contents of a DtEditor widget with the contents of a file
- DtEditorReset(3): Resets a DtEditor widget to its default state
- DtEditorSaveContentsToFile(3): Saves the contents of a DtEditor widget to a file
- DtEditorSelectAll(3): Selects all the text in a DtEditor widget
- DtEditorSetContents(3): Places data into a DtEditor widget
- DtEditorSetContentsFromFile(3): Loads data from a file into a DtEditor widget
- DtEditorSetInsertionPosition(3): Sets the position of the insert cursor in a DtEditor widget
- DtEditorTraverseToEditor(3): Sets keyboard traversal to the edit window of a DtEditor widget
- DtEditorUndoEdit(3): Undos the last edit made to the text in a DtEditor widget.

ComboBox Widget APIs. The DtComboBox widget is a combination of a TextField and a List widget that provides a list of valid choices for the TextField. Selecting an item from this list automatically fills in the TextField with that list item.

- #include <Dt/ComboBox.h>
- DtComboBox(5): DtComboBox widget definitions
- DtCreateComboBox(3): Creates a DtComboBox widget
- DtComboBox(3): DtComboBox widget class
- DtComboBoxAddItem(3): Adds an item to the ComboBox widget
- DtComboBoxDeletePos(3): Deletes a DtComboBox item
- DtComboBoxSelectItem(3): Selects a DtComboBox item
- DtComboBoxSetItem(3): Sets an item in the DtComboBox list.

MenuButton Widget APIs. The DtMenuButton widget is a command widget that provides the menu cascading functionality of an XmCascadeButton widget. DtMenuButton can only be instantiated outside a menu pane.

- #include <Dt/MenuButton.h>
- DtMenuButton(5): DtMenuButton widget definitions
- DtCreateMenuButton(3): Creates a DtMenuButton widget
- DtMenuButton(3): DtMenuButton widget class

SpinBox Widget APIs. The DtSpinBox widget is a user interface control for incrementing and decrementing an associated TextField. For example, it can be used to cycle through the months of the year or days of the month.

- #include <Dt/SpinBox.h>
- DtSpinBox(5): DtSpinBox widget definitions
- DtCreateSpinBox(3): Creates a DtSpinBox widget
- DtSpinBox(3): DtSpinBox widget class
- DtSpinBoxAddItem(3): Adds an item to the DtSpinBox
- DtSpinBoxDeletePos(3): Deletes a DtSpinBox item
- DtSpinBoxSetItem(3): Sets an item in the DtSpinBox list.

Calendar Library (libcsa)

Calendar APIs. The Calendar APIs include functions for inserting, deleting, and modifying entries, functions for browsing and finding entries, and functions for calendar administration.

- #include <csa/csa.h>
- csacsa(5): Calendar and appointment services definitions
- csa_add_calendar(3): Adds a calendar to the calendar service
- csa_add_entry(3): Adds an entry to the specified calendar

- csa_call_callbacks(3): Forces the invocation of the callback functions associated with the specified callback lists
- csa_delete_calendar(3): Deletes a calendar from the calendar service
- csa_delete_entry(3): Deletes an entry from a calendar
- csa_free(3): Frees memory allocated by the calendar service
- csa_free_time_search(3): Searches one or more calendars for available free time
- csa_list_calendar_attributes(3): Lists the names of the calendar attributes associated with a calendar
- csa_list_calendars(3): Lists the calendars supported by a calendar service
- csa_list_entries(3): Lists the calendar entries that match all the attribute search criteria
- csa_list_entry_attributes(3): Lists the names of the entry attributes associated with the specified entry
- csa_list_entry_sequence(3): Lists the recurring calendar entries that are associated with a calendar entry
- csa_logoff(3): Terminates a session with a calendar
- csa_logon(3): Logs on to the calendar service and establishes a session with a calendar
- csa_look_up(3): Looks up calendar information
- csa_query_configuration(3): Determines information about the installed CSA configuration
- csa_read_calendar_attributes(3): Reads and returns the calendar attribute values for a calendar
- csa_read_entry_attributes(3): Reads and returns the calendar entry attribute values for a specified calendar entry
- csa_read_next_reminder(3): Reads the next reminder of the given type in the specified calendar relative to a given time
- csa_register_callback(3): Registers the callback functions to be invoked when the specified type of update occurs in the calendar
- csa_restore(3): Restores calendar entries from an archive file
- csa_save(3): Saves calendar entries into an archive file
- csa_unregister_callback(3): Unregisters the specified callback functions
- csa_update_calendar_attributes(3): Updates the calendar attributes values for a calendar
- csa_update_entry_attributes(3): Updates the calendar entry attributes
- csa_x_process_updates(3): Invokes a calendar application's calendar event handler.

ToolTalk Messaging Library (libtt)

ToolTalk Messaging API. This API provides functions for managing all aspects of ToolTalk messaging.

- #include <Tt/tt_c.h>
- Tttt_c(5): ToolTalk messaging definitions.

ToolTalk Toolkit APIs. The ToolTalk toolkit APIs are a set of higher-level interfaces to the ToolTalk messaging APIs. The ToolTalk toolkit APIs facilitate use of the desktop message set and the media exchange message set.

- #include <Tt/tttk.h>
- Ttttk(5): ToolTalk toolkit definitions
- ttdt_Get_Modified(3): Asks if any ToolTalk client has changes pending on a file
- ttdt_Revert(3): Requests a ToolTalk client to revert a file
- ttdt_Save(3): Requests a ToolTalk client to save a file
- ttdt_close(3): Destroys a ToolTalk communication endpoint
- ttdt_file_event(3): Uses ToolTalk to announce an event about a file
- ttdt_file_join(3): Registers to observe ToolTalk events on a file
- ttdt_file_notice(3): Creates and sends a standard ToolTalk notice about a file
- ttdt_file_quit(3): Unregisters interest in ToolTalk events about a file
- ttdt_file_request(3): Creates and sends a standard ToolTalk request about a file
- ttdt_message_accept(3): Accepts a contract to handle a ToolTalk request
- ttdt_open(3): Creates a ToolTalk communication endpoint
- ttdt_sender_imprint_on(3): Acts like a child of the specified tool
- ttdt_session_join(3): Joins a ToolTalk session
- ttdt_session_quit(3): Quits a ToolTalk session
- ttdt_subcontract_manage(3): Manages an outstanding request.

Motif Toolkit Libraries (libXm, libMrm, libUil)

Motif Widget API. The CDE Motif Widget API (Xm) consists of the Motif 1.2 widget library (libXm) with enhancements to existing functionality and bug fixes. The CDE Motif widget API maintains source compatibility and binary compatibility with Motif 1.2 applications.

- #include <Xm/XmAll.h>

Motif Resource Manager API. The Motif resource manager API (Mrm) creates widgets based on definitions contained in user interface definition files created by the user interface language (UIL) compiler. The Motif resource manager interprets the output of the UIL compiler and generates the appropriate argument lists for widget creation functions.

- #include <Mrm/MrmAppl.h>
- #include <Mrm/MrmDecls.h>
- #include <Mrm/MrmPublic.h>

Motif User Interface Language (UIL) API. The Motif UIL is a specification language for describing the initial user interface of a Motif application.

- #include <uil/Uil.h>
- #include <uil/UilDBDef.h>
- #include <uil/UilSymDef.h>
- #include <uil/UilSymGl.h>

Accessing and Administering Applications in CDE

Setting up transparent access to applications and resources in a highly networked environment is made simpler by facilities that enable system administrators to integrate applications into the CDE desktop.

by Anna Ellendman and William R. Yoder

A major purpose of a graphical desktop is to make it easier for users to browse and run applications and to identify and manipulate the applications' data files. Since UNIX® systems are generally highly networked, it is desirable that a desktop also provide network transparency—the ability to launch applications and supply data to them without worrying about where in the network the applications and data are located.

Wherever possible, these ease-of-use features should not be provided at the expense of system administrators. There should be a standard procedure for incorporating preexisting applications into the desktop, and the desktop should provide tools for performing these procedures.

This article describes how users locate and launch applications from the CDE desktop and how system administrators integrate applications into the desktop's graphical environment. The information is also relevant for application developers, since the administration model and tools for integrating existing applications are also used to provide basic desktop integration for new applications.

User Model for Applications

CDE makes it easier for users to access and run applications by providing:

- A way to represent an application as an icon. The user can start an application by double-clicking the icon. These icons are called application icons.
- A special container for application icons. This container is called the *application manager*.
- A way to specify the unique appearance and behavior for icons representing an application's data files.

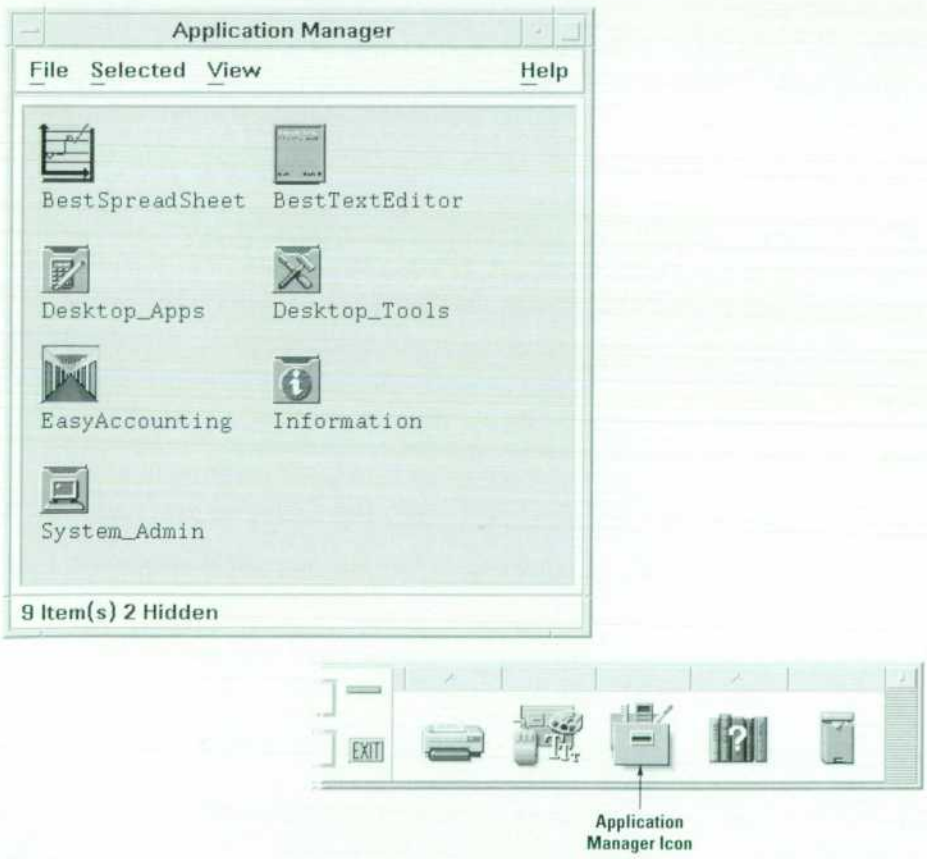


Fig. 1. CDE application manager window and the front panel icon for opening the window.

Application Manager and Application Icons. The application manager is a single container for all the applications available to the user and is opened by clicking a control in the CDE front panel (see Fig. 1). Each item in the top level of the application manager is a directory containing one or more related applications. These directories are called *application groups*.

By convention, an application group contains, for each application in the group, the application icon that starts the application, plus other files related to the application such as sample data files, templates, and online information (see Fig. 2). The system administrator can organize the application groups into deeper hierarchies.

Since the application groups are displayed together in a single window, they appear to occupy the same file system location. This makes applications easy to find and launch. The fact that this is not the case, and that the application groups are actually located in a variety of local and remote locations, is hidden from users.

From the end user's point of view, the application manager window is owned by the system. The user does not have the ability to create or move icons in the window directly.

Data Files and File Manager. Like the application manager, the *file manager* represents objects as icons. Frequently, these objects are the application data files. The desktop provides a way to specify the behavior of various types of data files. This makes it possible for users to start an application from the file manager by double-clicking one of its data files, by dropping a data file on the application icon, or by choosing Open from the data file's pop-up menu (see Fig. 3).

Fig. 2. Contents of an application group for a single application.

Application Manager Administration

The application manager is designed to meet several important requirements:

- It must appear to be a single location into which applications are gathered from a variety of locations.
- It must be customizable on a personal (per-user) or system-wide (per-workstation) basis.
- It must be network capable, that is, it must be able to gather applications located on other systems.
- It must be dynamic so that its contents can be changed when applications are added to or removed from the local system or application servers in the network.

To meet these requirements, the CDE designers chose to make the application manager a file manager view of a special temporary directory. The application manager directory is created automatically when the user logs in at the location:

```
/var/dt/appconfig/appmanager/login-display
```

For example, when user anna logs into CDE at display hpcvxpae:0, the CDE login manager creates the directory:

```
/var/dt/appconfig/appmanager/anna-hpcvxpae-0
```

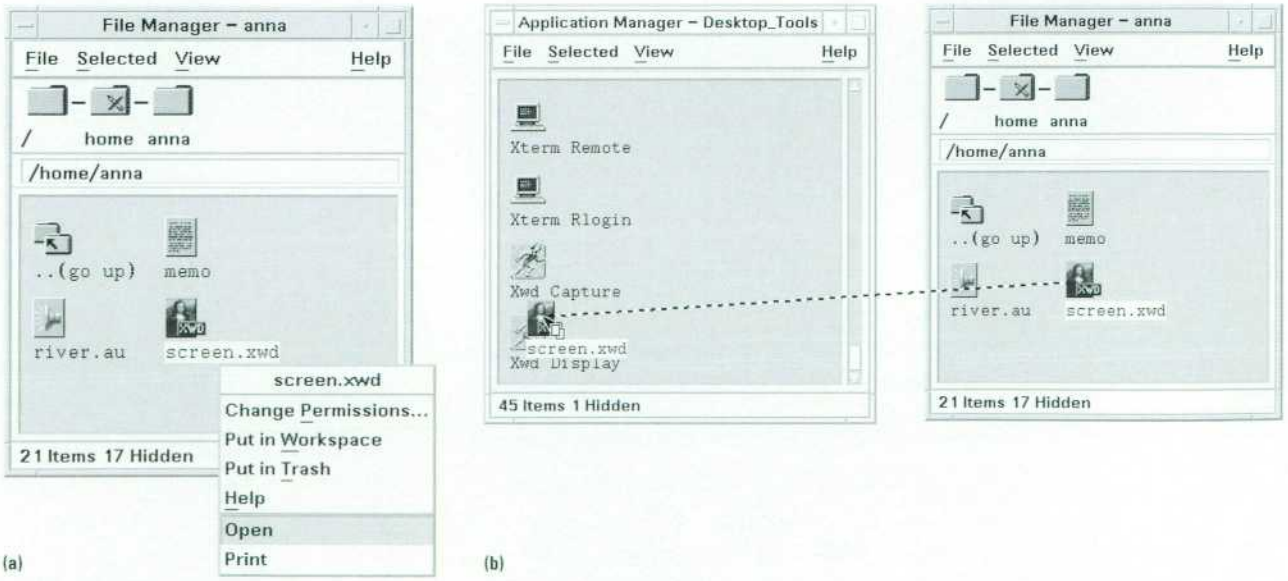


Fig. 3. Running an application using (a) a data file's pop-up menu in the file manager or (b) drag and drop between the file manager and the application manager.

The directory is both user and display dependent. Each user gets an application manager directory, and a user logging into more than one system obtains a separate application manager directory on each system. This is necessary to allow the application manager to be customized on both a per-user and a per-system basis.

The application manager directory persists for the length of the user's CDE session and is recreated each time the user logs in. The temporary nature of the application manager is possible because none of its contents are actually located in its file system location.

Gathering Applications. After the login manager creates the application manager directory, it must gather application groups into the directory. The application groups are gathered by means of symbolic links, and the links are made from multiple locations. This is what makes it possible for the application manager to display application groups located in a variety of directories, including personal, system-wide, and remote locations.

The symbolic links that gather the application groups are created by the CDE utility dtappgather, which is automatically run by the login manager each time the user logs in. For example, the desktop provides a built-in application group:

```
/usr/dt/appconfig/appmanager/C/Desktop_Apps
```

At login time, dtappgather creates the following symbolic link:

```
/var/dt/appconfig/appmanager/anna-hpcvxpae-0/
 Desktop_Apps->
/usr/dt/appconfig/appmanager/C/Desktop_Apps
```

The result is that the application manager contains the Desktop_Apps application group (see Fig. 4).

Application Search Path. To gather application groups from various locations, dtappgather requires a list of locations containing the application groups to be gathered. This list of locations is called the desktop's application search path.

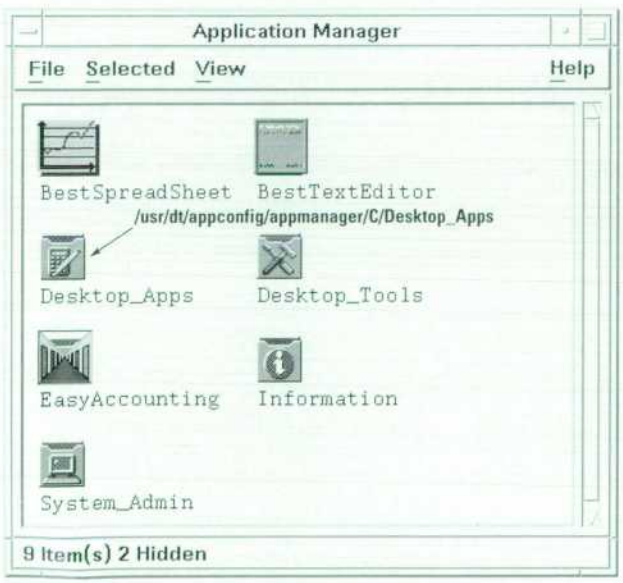


Fig. 4. The Desktop_Apps application group is a built-in group provided by the desktop.

The default application search path consists of three local locations:

Personal	`$HOME/.dt/appconfig/appmanager`
System-wide	`/etc/dt/appconfig/appmanager/<$LANG>`
Built-in	`/usr/dt/appconfig/appmanager/<$LANG>`

The built-in location is used for factory-installed application groups. System administrators can add application groups to the system-wide location to make those applications available to all users logging into the system. The personal location allows a user to add application groups that are available only to that single user.

System administrators or users may need to add other locations to the application search path. CDE provides two environment variables that modify the application search path: the system-wide variable DTSPSYSAPPHOSTS and the personal variable DTSPUSERAPPHOSTS.

The entire application search path, consisting of the default locations plus the additional locations specified by the environment variables, is created by a CDE utility named dtsearchpath. The login manager runs the dtsearchpath utility just before it runs dtappgather. The dtsearchpath utility uses a set of rules to define how the total value of the search path is assembled. Directories listed in the personal environment variable have precedence over the default personal location, and personal locations have precedence over system-wide locations.

The most common reason for modifying the application search path is to add remote locations on application servers so that those remote applications can be easily started by users. The application search path variables accept a special syntax that makes it easy to add application servers. For example, VARIABLE=hostname: is expanded by dtappgather (assuming NFS mounts) to the system-wide location on hostname:

```
/net/<hostname>/etc/dt/appconfig/appmanager/
<$LANG>
```

For example, if DTSPSYSAPPHOSTS specifies two remote systems:

```
DTSPSYSAPPHOSTS=SystemA:,SystemB:
```

and these systems contain the following application groups:

```
SystemA   /etc/dt/appconfig/appmanager/C/
           EasyAccounting
SystemB   /etc/dt/appconfig/appmanager/C/
          BestSpreadSheet
```

then dtappgather creates the following symbolic links:

```
/var/dt/appconfig/appmanager/anna-
hpcvxpae-0/EasyAccounting ->
/net/SystemA/etc/dt/appconfig/appmanager/C/
   EasyAccounting

/var/dt/appconfig/appmanager/anna-
hpcvxpae-0/BestSpreadSheet ->
/net/SystemB/etc/dt/appconfig/appmanager/C/
   BestSpreadSheet.
```

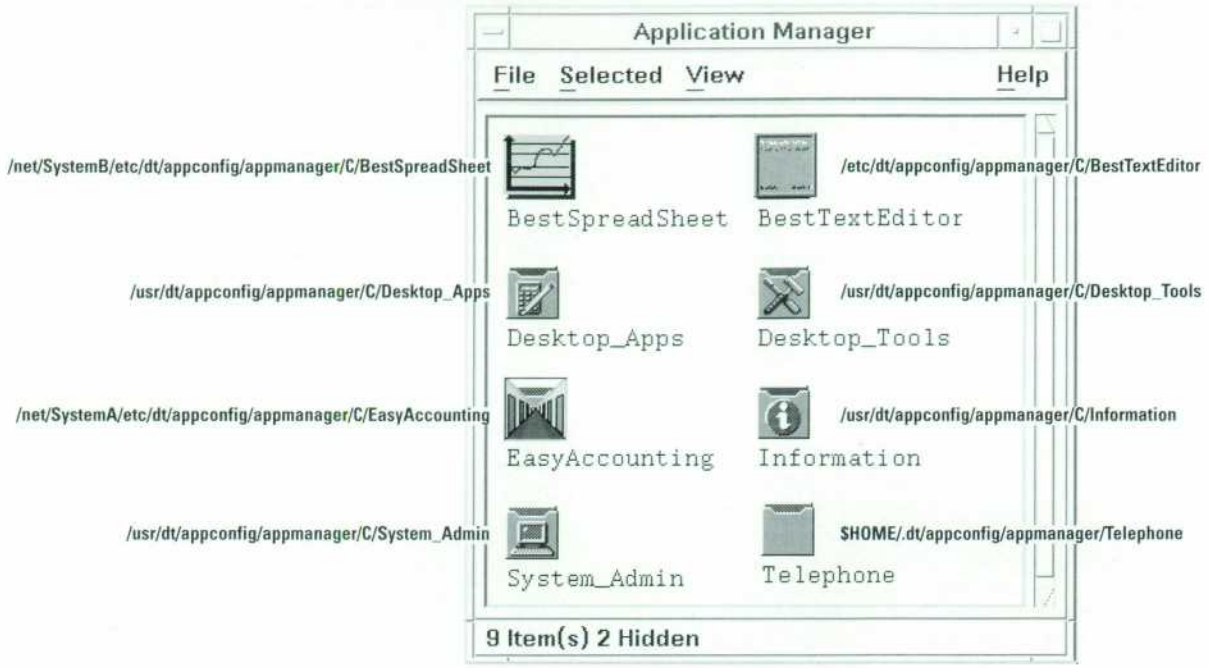


Fig. 5. The application manager gathers application groups on the application search path.

If the system uses a mount point other than /net, the system administrator can set a desktop environment variable DTMOUNTPOINT to specify the mount point location.

Fig. 5 shows an application manager containing application groups gathered from personal, system-wide, and built-in locations and from two application servers.

Application Infrastructure

The ability to represent applications and data files as icons that have meaningful behavior and appearance on the desktop is made possible by an infrastructure of desktop constructs and configuration files. These constructs are actions, data types, and icon image files.

Actions. The desktop uses actions to create a relationship between an icon in the application manager (or file manager) and a command. Actions are the constructs that allow you to create application icons (icons that the user can double-click to run applications).

For example, consider the following definition for an action named Xwud:

```
ACTION Xwud
{
   LABEL        Xwd_Display
   ICON         XwudIcon
   ARG_TYPE     XWD
   WINDOW_TYPE  NO_STDIO
   DESCRIPTION  Displays an X Windows screen\
                file
   EXEC_STRING   xwud -in %Arg_1"XWD file:"%
}
```

The desktop assembles and maintains a database of action definitions, including built-in actions and others created by users and system administrators. Once an action is defined in the actions database, it can be visually represented in the application manager or file manager as an icon. This icon is called an application icon because the underlying action usually is written to launch an application. Since icons in the file manager and the application manager represent files, creating an application icon involves creating a file that is related to the action. The relationship is provided by giving the file the same name as the action (in this case, Xwud), and by making the file executable. The real content of the file is irrelevant. Fig. 6 shows an application icon for the Xwud action.

The ICON and LABEL fields in the Xwud action definition describe the appearance—the icon image used and the text label for that icon. The DESCRIPTION and EXEC_STRING fields describe the icon's behavior. The DESCRIPTION field contains the text displayed in the CDE help viewer when the user selects the icon and requests help (**F1**). The EXEC_STRING specifies the command that is run when the user double-clicks the icon. For the Xwud action, the command is:

```
xwud -in <file>
```

where file is the file argument.

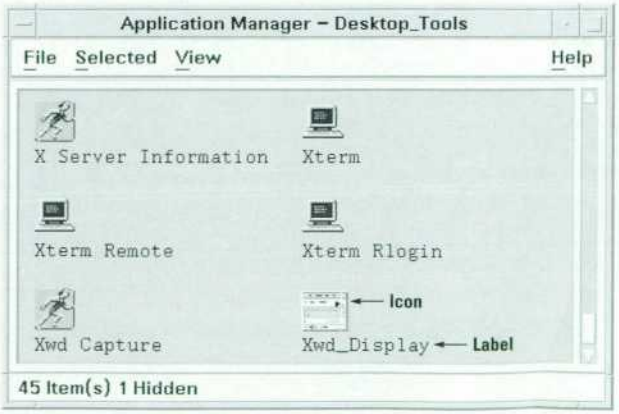


Fig. 6. Icon for the Xwud action.

The EXEC_STRING field uses a special syntax to represent file arguments. For example, the file argument in the Xwud action is represented as:

```
%Arg_1"XWD file:"%
```

This syntax specifies how the application icon treats data files. If the user drops a data file on the icon labeled Xwd_Display, its path name, supplied by the desktop drag and drop infrastructure, is used as the argument. If the user double-clicks the icon, the action prompts for that path by displaying a dialog box with a text field labeled XWD file:, which indicates that user must enter the file name of an X Window dump (.xwd) file.

The ARG_TYPE field limits the action to a particular type of data. If the user tries to use a different type of file, either by dropping the file on the action icon or responding to the prompt, the action returns an error dialog box.

Data Types. Files and directories are represented in the CDE file manager as icons that users can manipulate. There is little advantage to this iconic representation unless the desktop can supply meaningful appearance and behavior to these icons. Data typing provides this capability.

For example, directories appear in the file manager as folder-like icons, and users open a view of a directory by double-clicking its icon. That behavior, which is unique to directories, is provided by data typing.

The most common use for data typing is to provide a connection between data files and their applications. If an application reads and writes a special type of data, it is useful to create a data type for the data files. This data type might specify that the data files use a special icon image and that double-clicking a data file runs the application and loads the data file.

A data type definition has two parts:

- DATA_CRITERIA: specifies the criteria used to assign a file to that data type.
- DATA_ATTRIBUTES: defines a file's appearance and behavior.

For example, here is a data type definition for X Window dump files (the data files for the Xwud action):

```
DATA_CRITERIA XWD1
{
   DATA_ATTRIBUTES_NAME XWD
   MODE                 f
   NAME_PATTERN         *.xwd
}

DATA_ATTRIBUTES XWD
{
ACTIONS    Open,Print
ICON       Dtxwd
DESCRIPTION  This file contains  \
             an XWD graphic image.
}
```

The criteria definition specifies that the data type applies to files (MODE f) whose names (NAME_PATTERN) end with the suffix .xwd. (CDE also supports content-based data typing.)

The ACTIONS field in the attributes definition describes the file's behavior. The first entry (in this case, Open) describes the file's double-click behavior. All the entries in the ACTIONS list also appear in the file manager Selected menu (see Fig. 7).

In the desktop, Open and Print are general action names that are used with many data types. For .xwd files, the Open action is a synonym for the Xwud action, and the desktop must provide a connection between them. This connection is made through another action definition in which an action named Open is mapped to the Xwud action for the .xwd data type:

```
ACTION Open
{
TYPE          MAP
MAP_ACTION    Xwud
DATA_TYPE     XWD
}
```

When a user selects a data file in the file manager and runs the Open action on it (by double-clicking the file or by choosing Open from the Selected menu), the desktop searches for the appropriate Open action for that data type. Since this action is mapped to the Xwud action, the desktop then runs the Xwud action, and passes the selected data file to it as the file argument.

The Print action is handled similarly to Open. The following declaration is a set of actions for printing .xwd files:

```
ACTION Print
{
   TYPE          MAP
   MAP_ACTION    XWD_Print
   DATA_TYPE     XWD
}

ACTION XWD_Print
{
   ARG_TYPE    XWD
   EXEC_STRING  /bin/sh -c 'cat %(File)Arg_1%    \
                |xwd2sb|pcltrans -s -R -e2 >      \
                $HOME/temp.pcl;                   \
                dtaction PrintRaw $HOME/temp.pcl; \
                rm $HOME/temp.pcl'
{
```

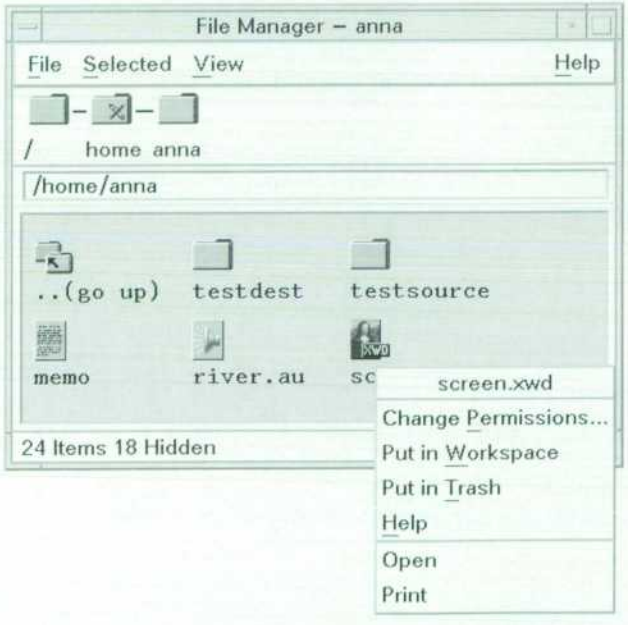


Fig. 7. An .xwd data file in the file manager, with Open and Print actions in its pop-up menu.

The XWD_Print action illustrates two additional features of actions:

- An action can invoke a shell (bin/sh in this case)
- An action can invoke another action. This is done using the dtaction command. The XWD_Print command changes the .xwd file to a PCL file and then invokes a built-in action named PrintRaw which prints PCL files.

The article on page 24 describes the different types of data structures and databases associated with CDE action and data types.

Icon Image Files. Since the desktop makes heavy use of icons to represent files, directories, and applications, icon image files are an important part of the suite of desktop configuration files.

A particular object on the desktop generally requires several icons. For example, the file manager has viewing preferences for representing files as large icons (32 by 32 pixels) or small icons (16 by 16 pixels). Furthermore, the desktop uses pixmaps for color displays and bitmaps for monochrome displays. To differentiate icons by size and type (pixmap or bitmap), icon files use the naming convention basename.size.type. For example, a large pixmap for the Xwud action might be named Xwud.l.pm.

The icon image used by an object is specified in its action or data type definitions by the ICON field (see examples above), which specifies the icon image to use in the file manager and the application manager. The ICON field specifies only the base name, and the desktop adds the appropriate file name extensions, depending on the file manager view setting and type of display. Furthermore, the ICON field does not specify a path. The desktop uses search paths to locate icons and other parts of the desktop infrastructure.

Locating Actions and Icons. As with application groups, the desktop uses search paths to locate action and data type definitions and icon files. Each application search path location has a set of corresponding locations for actions, data types, and icons. For example, Fig. 8 shows the default system-wide search path locations. The types directory for actions and data types and the icons directory for icon image files are analogous to the appmanager directory for the application groups.

The help directory is used for application help files created using the CDE Help Developer's Kit. CDE help is described on page 38.

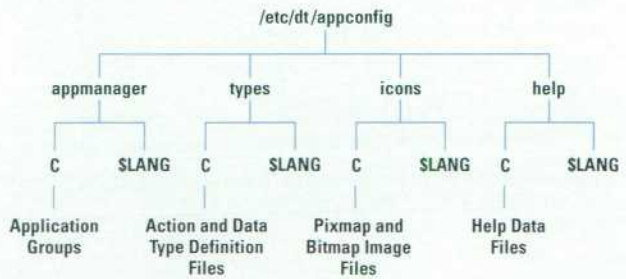


Fig. 8. Directory structure of system-wide desktop configuration files.

The search paths for action, data type, icon, and help files are automatically updated when the application search path is modified. This ensures that the desktop will find all the desktop configuration files needed by the application.

For example, if SystemA: is added to the application search path, the locations:

```
/net/SystemA/etc/dt/appconfig/types/<$LANG>
/net/SystemA/etc/dt/appconfig/icons/<$LANG>
/net/SystemA/etc/dt/appconfig/help/<$LANG>
```

are automatically added to the action/data type, icon, and help search paths.

Create Action

The syntax of action and data-type definitions provides a great deal of flexibility in specifying desktop behavior. While this makes the definitions very powerful, it also makes the syntax somewhat complex.

The desktop includes an application, *create action*, that allows users and system administrators to create typical actions and data types without having to learn the syntax rules for the definitions. Create action provides fill-in-the-blank text fields for supplying various parts of the action and data type definitions and provides a way to browse and choose from available icons (see Fig. 9). Furthermore, create action allows the user to enter the command to be executed (EXEC_STRING) using shell language for file arguments (e.g., $n rather than %(File)Arg_n%).

Create action is optimized for creating actions and data types for applications. It creates an action to launch the application and one or more data types for the application. For each data type, create action produces an Open command that runs the application. Optionally, it can also create a Print action.

Application Integration

An application can be integrated into the desktop by placing its desktop configuration files into the locations specified by the desktop search paths shown in Fig. 8. However, this can make administration difficult because the files are scattered among numerous directories, each of which contains files for many applications.

It is usually preferable to gather all the configuration files for an application in one location, called the application root (app_root). Applications generally are installed this way. For example, the files for an audio application might be installed under /opt/audio. However, since the directories under the app_root are not on the desktop search paths, the application groups, actions, data types, and icons would not be found unless the search paths were modified.

Application Registration. CDE provides the utility dtappintegrate which lets system administrators install desktop-related application files under an app_root directory without modifying the desktop search paths. The function of dtappintegrate is to create links between the desktop files in the app_root location and the system-wide search path locations. The process of creating these links with dtappintegrate is called application registration.

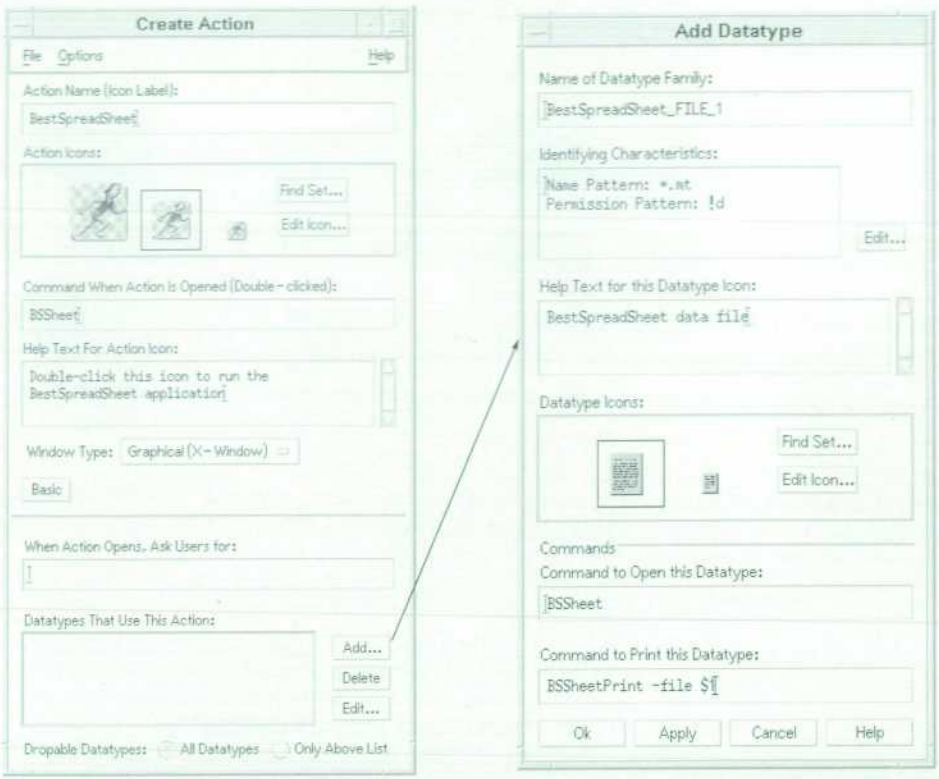


Fig. 9. Create action main window and dialog box.

The collection of desktop configuration files beneath the app_root directory is called the registration package. Fig. 10 illustrates a registration package for an application named BestSpreadSheet.

The registration package contains the desktop configuration files needed by the application, including the application group, actions, data types, icons, and help files. The registration package uses the same directory organization as the search paths (compare Fig. 10 with Fig. 8).

Once the registration package has been created, the registration is accomplished by running the dtappintegrate utility, which creates the symbolic links shown in Fig. 11.

The system administrator can also use dtappintegrate to break the symbolic links for an application's desktop configuration files. This is necessary to move an application to a different application server. In addition, the system administrator can also use dtappintegrate to move the application configuration registry to a different location on the application server to

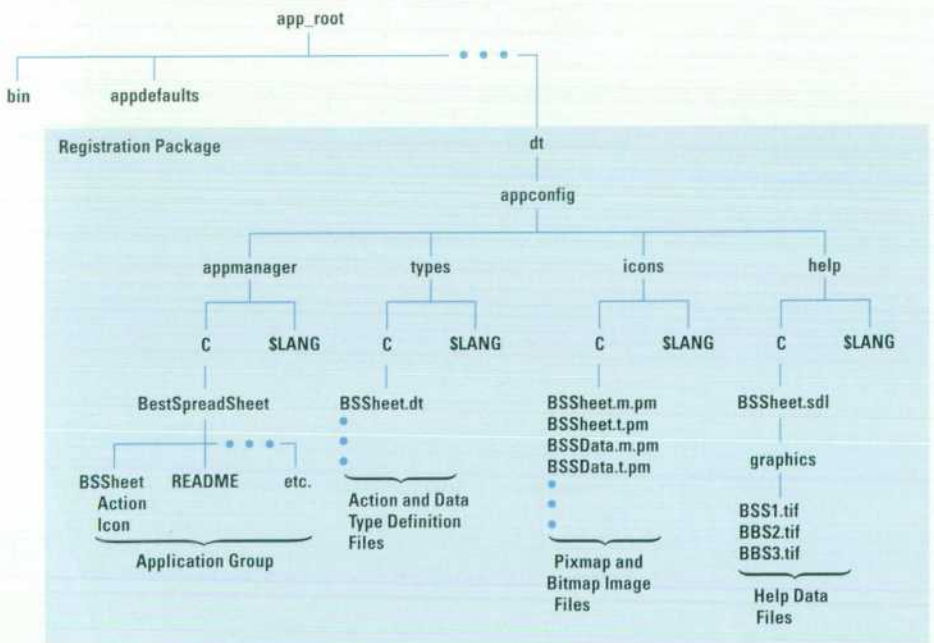


Fig. 10. Example of a registration package.

Application Servers and Clients in CDE

CDE operates in a networked, client/server environment. This article describes the services and configuration files provided in CDE to support client/server remote execution.

CDE Networking Services

The CDE networking model includes four primary services:

- **Desktop display services.** These services consist of the X11 display server, keyboard and pointing devices, and limited auxiliary processing, such as window management and audio services.
- **Session services.** Session servers provide login and authentication, session management, and desktop services such as file manager, style manager, and action invocation.
- **Application services.** Application servers provide both terminal and GUI-based applications. When an application is located and running on a system other than the session server, it is said to be a remote application, and the system running the application is the application server.
- **File services.** File servers store the data that applications produce and manipulate.

The primary way that application servers, file servers, and session servers share data in CDE 1.0 is through the remote file system (RFS) mechanisms, as offered by the DCE Distributed File Service (DFS),[1] the Andrew File System (AFS), and the Network File System (NFS).

There may be other system services available in the distributed desktop environment, such as font, printing, and fax servers.

Remote Application Models

The desktop can access a remote application by RFS-based execution or by remote application execution.

- **RFS-based execution.** In this configuration, the application binaries reside on the remote system but are run on the session server. From the session server's point of view, the remote system is simply a big disk attached to the workstation by means of RFS. The remote system is not a true CDE application server because it is not providing application execution services.
- **Remote application execution.** In this configuration, the application runs on the application server and displays its output on the user's desktop display. This configuration requires the linkages shown in Fig. 1. An advantage of this configuration is that the application server can be a different machine type or operating system version than the session server. This is common in today's heterogeneous environments.

CDE provides a small subprocess control daemon (dtspcd) that is installed on application servers. The subprocess control daemon receives requests from CDE components (such as the application manager) to launch processes on the application server on behalf of the user.

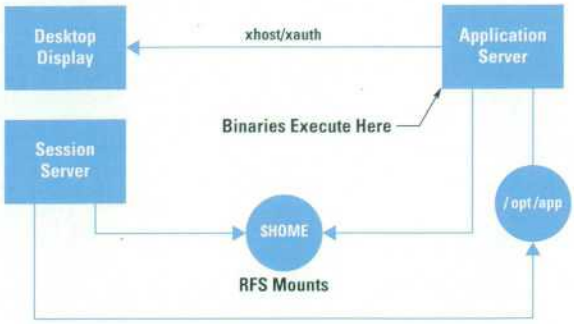


Fig. 1. Linkages for remote application execution.

Configuring a CDE Application Server

Typically, a CDE application server contains the application binaries, application configuration files (such as app-defaults and message catalogs), and the application's CDE desktop configuration files (application group, action and data type definition files, icon image files, and help files). Locating all the application's configuration files on one system makes the application easier to install and administer. During or after installation, the dtappintegrate utility is run to link the configuration files to a location where clients expect to find them. In addition, the subprocess control daemon must be configured, and clients must have permission to access files by RFS.

Configuring a CDE Application Client

A system accessing a CDE application server must add the application server to its application search path. This is done by setting a system-wide or personal environment variable. For example, the following line in the file /usr/dt/config/Xsession.d/0010.dtpaths adds an application server named SystemA to the client's application search path: DTSPSYSAPPHOSTS=SystemA:.

In addition, the client must have RFS access to the application server. Furthermore, a client must be configured to provide X authorization for the application server to access the client's display. The easiest way to set up authorization is to configure the user's home directory so that it is available to all application servers. This configuration is called a networked home directory.

Reference

1. M. Kong, "DCE: An Environment for Secure Client/Server Computing," Hewlett-Packard Journal, Vol. 46, no. 6, December 1995, pp. 6-22.

control which users can access an application from the desktop.

Regathering Applications. Applications registered by dtappintegrate do not become available from the application manager until the utility that gathers applications from search path locations, dtappgather, is rerun. Since dtappgather runs automatically at login time, newly registered applications become available when the user logs out and back in again. To save users the inconvenience of logging out, the desktop provides an action named Reload Applications. The user can run this action by opening the application manager and double-clicking the Reload Applications icon.

Conclusion

CDE provides the ability to represent applications and their data files as icons, and supplies a single container for applications called the application manager. Applications are gathered into the application manager from multiple locations specified by the application search path. Remote locations can be added to the application search path so that

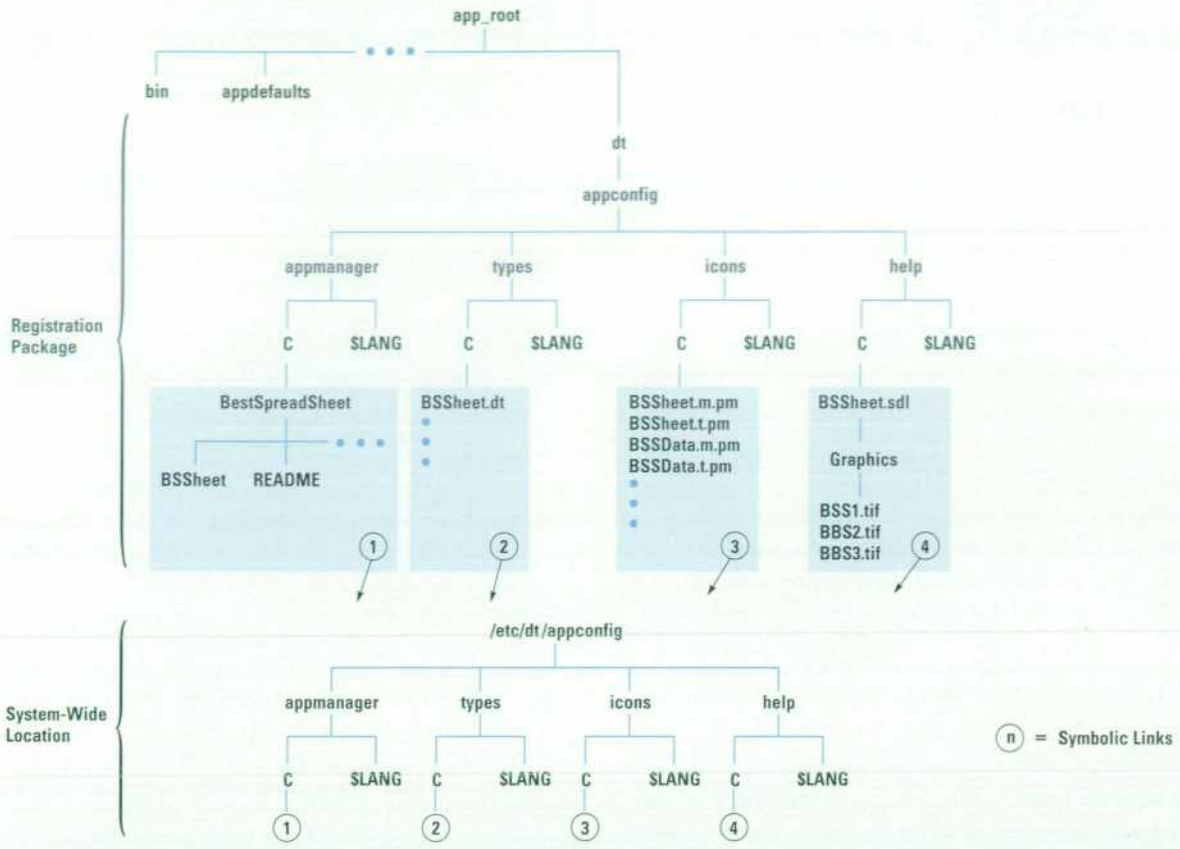


Fig. 11. Links created by dtappintegrate.

applications on networked application servers can be integrated seamlessly into the application manager. The infrastructure required to represent applications as icons consists of actions, data types, and icon image files. These files can be placed into a registration package for the application and then registered onto the desktop using the dtappintegrate utility.

Acknowledgments

Thanks to Jay Lundell, Bob Miller, and Brian Cripe for contributing to the design of the application administration model for CDE and to Julia Blackmore for her great attention to detail in reviewing the documentation. Special thanks to John Bertani for his work on the search path utilities and to Ron Voll for his work on actions. Finally, we were fortunate to have the opportunity to work with Troy Cline and Sandy Amin at IBM as they implemented dtappintegrate and the file manager and with Rich McAllister of SunSoft who contributed the CDE file mapping routines.

The CDE Action and Data Typing Services

Several different types of databases and their associated APIs are involved in determining the look and behavior of icons presented in the Common Desktop Environment.

by Arthur F. Barstow

Two fundamental requirements of a computerized desktop system are a unified view of a user's data and a consistent access method to the data. This article describes how Hewlett-Packard's Common Desktop Environment (CDE) meets these requirements through the use of its data typing and action services. The data typing service defines attributes of a data type such as its appearance (icon) and behavior (action). The action service provides mechanisms for linking a data type's behavior to its associated application or execution command.

The data typing service and the action service both use databases. The data typing service database contains data criteria and data attribute records, and the action service database contains action records. The term database in the context of data typing and action databases refers to records stored in flat, ASCII files and not in a generalized database service.

Applications wanting to use these CDE services must first load the databases into the application's memory. Once loaded, application programming interfaces (APIs) described in this article can be used to access the database. CDE only provides APIs to retrieve data from the databases.

Each database record consists of the record type name, the name of the record, and one or more attributes (also called fields). Each attribute has a name and a value and each attribute must begin on a separate line. Fig. 1 shows the basic elements of a database record.

Although CDE defines numerous data types and actions in its default databases, these databases can be augmented with system-wide and user-defined records. User-defined definitions have the highest precedence, followed by systemwide definitions. The default databases have the lowest precedence.

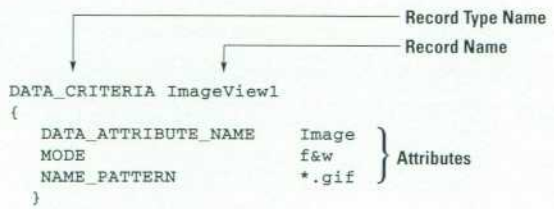


Fig. 1. The basic structure of records contained in the CDE data typing and action databases.

Fig. 2 shows an overview of the CDE data typing and action services and and their interrelationships. These data structures and their relationships are described in this article.

The Data Typing Service

As mentioned above, the data typing service contains data criteria and data attribute records. Data criteria records contain rules for recognizing a data type. The rules may specify a data type's file name extension or file permissions. Data attribute records are used to define a data type's appearance and behavior such as a type's icon and associated applications. A data type can be defined for a file or a buffer of memory.

Data Criteria Records. Data criteria records define the rules for recognizing or identifying a data type by using the following criteria:

- File name (including shell pattern matching symbols)
- Contents (may be a file or memory buffer)
- File permissions and file type (a file type may be a directory, symbolic link, etc.).

The following declaration contains three data criteria records for an image viewer application that can display files and data in formats such as X11 bitmaps, GIF images, and X11 pixmaps.

```
DATA_CRITERIA ImageView1
{
   DATA_ATTRIBUTE_NAME Image
   MODE                f&w
   NAME_PATTERN        *.gif
}

DATA_CRITERIA ImageView2
{
   DATA_ATTRIBUTE_NAME Image
   PATH_PATTERN        */bitmaps/*.bm
}

DATA_CRITERIA ImageView3
{
   DATA_ATTRIBUTE_NAME Image
   CONTENT             0 string #define
}
```

All of these records refer to the data type Image. The ImageView1 record will match any writable file with a file name extension of .gif. The ImageView2 record will match any file

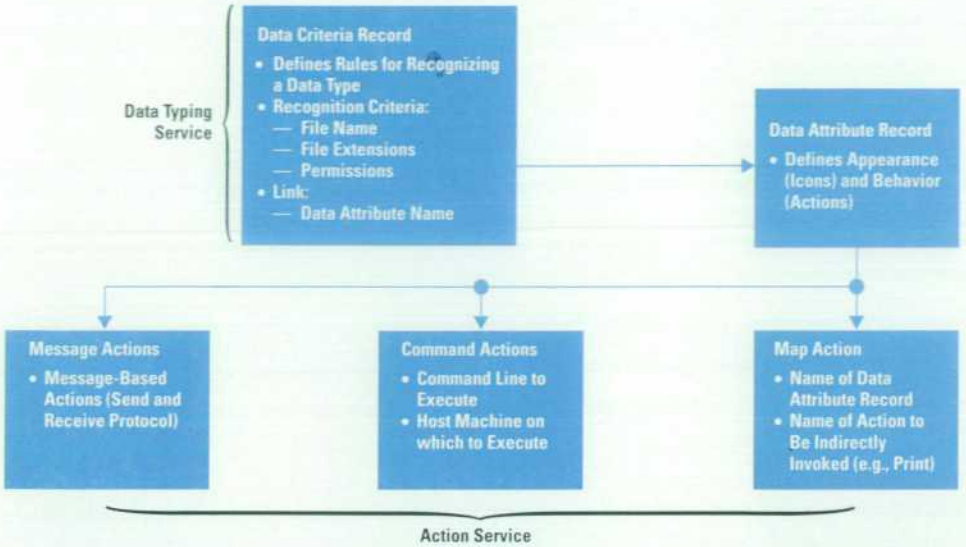


Fig. 2. An overview of the relationships between data typing and action service data strucures.

ending in .bm, but the file must be in a directory named bitmaps. The ImageView3 record will match any file containing the string #define at the beginning of the file. Fig. 3 contains a pseudo-BNF (Backus-Nauer form) for a data criteria record.

Data Attribute Records. Data attribute records define a data type's name as well as other attributes of the data type such as its icon and application binding. The data type name is the same as the data attribute record name. The following declaration shows the data attribute record for the Image data attribute defined in the data criteria records above. A data attribute record can be associated with an unlimited number of data criteria records.

```
DATA_ATTRIBUTE Image
{
   ACTIONS          Open, Print
   ICON             imagedata
   MIME_TYPE        image/jpeg
   PROPERTIES       visible
   MEDIA            Image_Data
   DESCRIPTION      Data type for the
                    ImageViewer application
}
```

This data type's icon is imagedata. Applications such as the CDE file manager will use this icon to represent image files. For example, if the file manager finds a file named kite.gif, the file will be represented with the imagedata icon. This data type also defines attributes that may be of interest to other data-typing-aware applications. For example, a mailer application may use the value of the MIME_TYPE attribute to decide how an attachment should be viewed.

Data attribute records are the only record type that can contain application-specific fields—they are not limited to a fixed set of field names. Fig. 4 shows a pseudo-BNF for data attribute records.

Data Typing Service APIs. Before the data typing APIs can be used, an application must first load the databases by calling the DtDbLoad function. After this, an application should register a database modification callback function using the DtDbReloadNotify function. This callback will be invoked when a user invokes the action ReloadActions to notify data-typing-aware applications that a data type or action record has been added, modified, or deleted. If an application fails to register the modification callback, it will not be notified of a database change, resulting in the application having an outdated database in memory and possibly appearing and behaving differently than applications that received the notification.

The first consideration in choosing the API to retrieve information from the data type database is whether the application already has a data type name for an object or if the object of interest is a file or a pointer to a memory buffer. For example, if an application wants to determine the icon associated with the file tanager.gif, it would first make the following call:

```
char *data_type_name = DtDtsFileToDataType
                       ("tanager.gif")
```

to determine the data type name of the file. To retrieve the icon for this data type, the application would then call:

```
char *icon_name =
    DtDtsdatatypeToAttributeValue
                    (data_type_name,
                    "ICON",
                    NULL)
```

For the data criteria and data attribute records given above, these two sequences of calls would result in setting icon_name to imagedata.

The next consideration is whether the desired information is a specific attribute of a data type or a (NULL-terminated) list of all attribute name and value pairs for a data type. The following function retrieves all of the attributes for a memory buffer:

```
DtDtsAttribute **attributes =
                DtDtsBufferToAttributeList
                    (void *buffer,
                    int buffer_length,
                    NULL)
```

For performance reasons, if an application needs more than one attribute for a data type, the APIs to retrieve all attributes with a single call should be used.

The Action Service

In CDE, the action service provides the infrastructure for consistent data access through polymorphic action naming.

```
DataCriteriaDefn   ::= 'DATA_CRITERIA' blanks record_name
'{'
  data_criteria_defn
'}'

data_criteria_defn ::= (
    'PATH_PATTERN' blanks pattern_datas newline
|   'NAME_PATTERN' blanks pattern_datas newline
|   'LINK_PATH' blanks pattern_datas newline
|   'LINK_NAME' blanks pattern_datas newline
|   'CONTENT' blanks content_fields newline
|   'MODE' blanks mode_specs newline
|   'DATA_ATTRIBUTES_NAME' blanks name newline
    )
 pattern_datas      ::= pattern_data [('&' | '|') pattern_datas]
 pattern_data       ::= ['!'] pattern
 pattern            ::= a shell pattern matching expression, as
   defined in sh(1)
 mode_specs         ::= mode_spec [('&' | '|') mode_specs]
 mode_spec          ::= (
   type_spec [permission_spec]
|   type_spec ('&' | '|') permission_spec
   )
 type_spec          ::= ['!'] type_char {type_char}
 type_char          ::= ('d' | 'l' | 'f' | 's' | 'b' | 'c' )
 permission_spec    ::= ['!'] permission_char {permission_char}
 permission_char    ::= ('r' | 'w' | 'x')
 content_fields     ::= content_field [('&' | '|') content_fields]
 content_field      ::= (
   ['!'] offset blanks 'string' blanks string
|  ['!'] offset blanks 'byte'  blanks data_values
|  ['!'] offset blanks 'short' blanks data_values
|  ['!'] offset blanks 'long'  blanks data_values
|  ['!'] offset blanks 'filename' blanks string
   )
 offset             ::= an unsigned decimal integer
 data_values        ::= data_value [blanks data_values]
 data_value         ::= an unsigned integer: decimal, octal (leading
   0) or hexadecimal (leading 0x or 0X)
 name               ::= ( "A-Z" | "a-z" ) [name_char]
 name_char          ::= { "A-Z" | "a-z" | "0-9" | "-" }
 string             ::= a character string, not including <newline>
 newline            ::= '\n'
 blanks             ::= one or more <blank>s (spaces and/or tabs)
```

Fig. 3. The pseudo-BNF for a data criteria record.

Desktop components, such as the CDE file manager and the CDE mailer, use the action service to start an application with a user's data. When a user opens a data file in the file manager, the action service opens the application associated with the data type. For example, when a mail folder is dropped on the mail control in the CDE front panel, the action service is used to open a mail instance with the dropped folder.

The following action types are provided in the action service:

- Map actions, which provide consistent action naming
- Command actions, which encapsulate command lines
- Message actions, which use the CDE message service to encapsulate a request or notification message.

```
DataAttributesDefn   ::= 'DATA_ATTRIBUTES' blanks record_name
'{'
                         data_attributes_defn
'}'

data_attributes_defn ::= (
                          'DESCRIPTION' blanks string newline
                      |   'ICON' blanks string newline
                      |   'INSTANCE_ICON' blanks string newline
                      |   'PROPERTIES' blanks string {',' string} newline
                      |   'ACTIONS' blanks name {',' name} newline
                      |   'NAME_TEMPLATE' blanks string newline
                      |   'IS_EXECUTABLE' blanks string newline
                      |   'MOVE_TO_ACTION' blanks string newline
                      |   'COPY_TO_ACTION' blanks string newline
                      |   'LINK_TO_ACTION' blanks string newline
                      |   'IS_TEXT' blanks string newline
                      |   'MEDIA' blanks string newline
                      |   'MIME_TYPE' blanks string newline
                      |   'X400_TYPE' blanks string newline
                      |   unique_string blanks string newline
                      |   '#' string newline
                          )
string               ::= a character string, not including <newline>
newline              ::= '\n'
unique_string        ::= a uniquely named string for implementation
                         extensions
blanks               ::= one or more <blank>s (spaces and/or tabs)
```

Fig. 4. The pseudo-BNF for a data attribute record.

Map Actions. Map actions provide a mechanism to present action names consistently, regardless of data type. CDE provides numerous data types and each type has an associated open action (an open action usually starts the type's application). To get consistent naming, the open action for each data type is named Open and each data type's print action is named Print. This allows applications such as the file manager to include in a data file's list of actions the names Open and Print, regardless of the actual action used to open or print the data type. The following are examples of map actions.

```
ACTION Open
{
   TYPE          MAP
   ARG_TYPE      Image
   MAP_ACTION    Open_Image
}

ACTION Print
{
   TYPE          MAP
   ARG_TYPE      Image
   MAP_ACTION    Print_Image
}

ACTION Open
{
   TYPE          MAP
   ARG_TYPE      Gif
   MAP_ACTION    Open_Gif
}
```

The TYPE attribute is MAP for map-type actions. The ARG_TYPE attribute gives the name of the associated data attributes record. The MAP_ACTION attribute gives the name of the action that will be indirectly invoked when the Open action is invoked.

The first Open and Print definitions are map actions for the Image data type defined in the Image data attribute record given above. Data attribute records and action records are linked by an action's ARG_TYPE field. If an application invokes an open action on an Image data type, the resulting action that gets invoked is Open_Image.

The second Open definition is another open action but for GIF-type files. When data of this type is opened, the Open_Gif action is indirectly invoked. Overloading map action names makes it possible to present the user with consistent action names.

Command Actions. Command actions are used to encapsulate arbitrary command lines (e.g., UNIX® commands and shell scripts) and application execution strings. Command actions, identified with a TYPE field of COMMAND, have three types of attributes: invocation, signature, and presentation. The following declaration shows these three types with a command action definition for the Open_Image action.

```
ACTION Open_Image
{
 # Invocation attributes:
 TYPE          COMMAND
 EXEC_STRING   /opt/imageviewer\
               bin/imageviewer\
               %Arg_1%
 WINDOW_TYPE   NO_STDIO
 EXEC_HOST     %DatabaseHost%,\
               mothra.x.org

 # Presentation attributes:
 ICON          imageapp
 LABEL         Image Viewe
 DESCRIPTION   Invokes the Image-\
               viewer application\
               for Image data types

 # Signature attributes:
 ARG_TYPE      Image
 ARG_COUNT     *
 ARG_CLASS     *
}
```

The invocation attributes define an action's execution parameters. The EXEC_STRING attribute contains the action's command line. The command line may contain keywords (e.g., %Arg_1%) which are replaced when an action is invoked.

The WINDOW_TYPE attribute specifies the window support required by the action. In this example, the image viewer application creates its own window so its WINDOW_TYPE is NO_STDIO. Other window types are available for commands needing terminal emulation support.

The EXEC_HOST attribute is used to specify a list of potential hosts on which the command could be executed. A keyword can be used to refer to a host name generically rather than explicitly naming a host. For example, the keyword %DatabaseHost% refers to the host on which the action is defined. When a list of hosts is used, the command will only be executed once, on the first host on which the command exists.

The presentation attributes (all are optional) are ICON, LABEL, and DESCRIPTION. Applications like the file manager use the ICON attribute to determine the icon to use to represent an action. The LABEL attribute defines a (potentially localized) user-visible name for an action. The DESCRIPTION attribute contains a short description of the action. An application should display the description as a result of a user's request for specific information about an action.

The signature attributes ARG_TYPE, ARG_COUNT and ARG_CLASS define the arguments accepted by an action. ARG_TYPE specifies a list of the data type names an action accepts. The data type names refer to data attribute record names. The ARG_COUNT attribute specifies the number of arguments an action accepts. The ARG_CLASS attribute specifies the class of arguments the action accepts, such as files or memory buffers.

When an application invokes an action, it gives the action service an action name and data arguments. The action service first searches the action database for actions matching the action name. Next the action record that matches the action name is checked for a match between the action record's signature attributes and the data arguments. If a match is found, the associated action will be invoked, otherwise an error is returned to the application.

Message Actions. The CDE message service provides the ability for applications to send and receive messages. The action service in turn provides the ability to encapsulate a message. A message-type action is specified by setting the TYPE attribute to TT_MSG. The following is an example of a message-type action.

```
ACTION OpenDirectoryView
{
 TYPE            TT_MSG
 TT_CLASS        TT_REQUEST
 TT_SCOPE        TT_SESSION
 TT_OPERATION    Edit
 TT_FILE         %Arg_1"Folder to open:"%
 TT_ARG0_MODE    TT_INOUT
 TT_ARG0_VTYPE   FILE_NAME
 DESCRIPTION     Request the File\
                 Manager to open a \
                 user-specified folder
}
```

The TT_CLASS attribute specifies if the message is a request or a notification. A request message requires that the application that accepts the message respond to the request. A notification message may be accepted by more than one application but no response is required.

The TT_SCOPE attribute specifies the message's scope. The scope of a message is typically the user's current CDE session. However, a message can also be file-scoped. File scoping is used by applications that are interested in receiving message events for a specific file. For example, if a file-scoping-aware editor opens or saves a file, other applications that are also file-scoped to the same file will be notified of the operation.

The TT_OPERATION attribute specifies the message's operation. The operation field may be application-specific, but may also specify a generic desktop operation such as Display or Edit as defined by the message service's media exchange message set.

The TT_FILE attribute specifies a file name for a message. In the example above, this field contains a keyword that results in the user being prompted for the name of a folder to open. The user's input will be placed in the message before it is sent.

A message can have any number of arguments. The arguments can be used to give data to the servicing application or to specify that data should be returned to the requesting application. The attributes TT_ARGi_MODE and TT_ARGi_VTYPE specify the input/output mode and type of a message for the ith message argument.

Actions, Data Typing, and Drag and Drop. The data typing and action services can be combined to define the drag and drop semantics for a data type. The following data attribute definition defines its default action (the action to be invoked when data of this type is opened) as OpenAction. This definition uses the MOVE_TO_ACTION, COPY_TO_ACTION, and LINK_TO_ACTION attributes to define actions to be invoked when data of this type is moved, copied, and linked respectively.

```
DATA_ATTRIBUTE DragAndDropAwareType
{
   ACTIONS           OpenAction

   MOVE_TO_ACTION    MoveAction
   # Move = Shift + Mouse Button 1

   COPY_TO_ACTION    CopyAction
   # Copy = Control + Mouse Button 1

   LINK_TO_ACTION    LinkAction
   # Link = Control + Shift + Mouse Button 1
}
```

Action Service APIs. The function used to invoke an application, DtActionInvoke, has several arguments. However, the parameters described here include the name of the action to invoke, an array of data arguments for the action, the number of arguments, and a callback function to be invoked when the action terminates.

The action name is used to find action records with the same name in the action database, and the signature attributes of the matching action records are searched to find a match with the API's data arguments. If a matching action is found, the action is invoked and an action invocation identification number is returned to the application, otherwise an error is returned to the application. The data arguments can be files or memory buffers. If an action is defined to accept zero or one arguments but more than one argument is provided, an instance of the action will be invoked for each argument.

To be notified when an action terminates, an application can register an action termination callback when the action is invoked. This is particularly useful for applications that invoke actions on behalf of embedded objects. For example, the mailer uses this feature to get notified when its attachments, which have been opened by an action invocation, are closed. If an action has more than one action instance outstanding, it can use the action invocation identification number to determine which action instance terminated.

When a data argument is a memory buffer, the application must provide a pointer to the buffer, the size of the buffer, and an indication of whether the action is allowed to modify the buffer. Additionally, the application can provide the data type of the buffer and a file name template to be used if a temporary file for the buffer is needed. When necessary, such as when a buffer is writable, the buffer is copied to a temporary file and the file name is used in the action invocation. When the action terminates, the contents of the file are copied to the buffer, and the temporary file is removed.

Migrating HP VUE Desktop Customizations to CDE

With CDE becoming the UNIX® desktop standard, it is important to allow HP VUE users to preserve their customizations when moving over to the CDE desktop. A set of tools has been developed to make this transition as complete and effortless as possible.

by Molly Joy

The HP Visual User Environment (HP VUE), combined with a set of applications, is a powerful graphical environment for interacting with the computer. It provides a consistent user interface and is based on OSF/Motif, which was created to enable different applications to behave the same way, eliminating the need for users to learn multiple command sets to control applications.

The need for standards was recognized early by Hewlett-Packard with its support for industry standards such as the X Consortium and the Open Software Foundation (OSF). Although HP VUE provided users with an easy-to-use desktop interface, there was no industry standard graphical user interface (GUI) for desktop computers running the UNIX operating system and the X Window System. What this meant was that even though Motif applications behaved the same across multiple platforms there was no commonality in the graphical user interface, which was referred to as a desktop. This was a serious limitation from the perspective of both the end user who had to learn to operate different desktops in a heterogeneous computing environment and the application developer who had to develop and integrate applications into multiple desktops. This was also a cost factor in enterprises with multivendor computing environments because of the costs involved in training users and integrating new systems into existing heterogeneous networked environments.

Hewlett-Packard has a long-standing commitment to open system standards. HP is one of four sponsor companies that contributed key technologies and helped to develop CDE (Common Desktop Environment). CDE's consistent look and feel is based on HP's proven and accepted HP VUE technology. This rich graphical user interface has become a core component of CDE.

Although HP VUE and CDE have the same look and feel, the two technologies are somewhat different with the result being that HP VUE customizations cannot be directly incorporated into the CDE desktop and be expected to work. HP's commitment to supporting customer investments dictated that a seamless transition from HP VUE to CDE was necessary. Even though a complete transition is not possible in some cases, the intent was to make the transition as complete and effortless as possible.

Developing the Migration Tools

The decision to develop a set of tools to allow HP VUE-to-CDE migration was made in the second half of 1994. This decision included an agreement between Hewlett-Packard and TriTeal Corporation to develop these tools jointly. TriTeal and HP have a long history with the HP VUE product because through a licensing agreement with HP, TriTeal has marketed and sold HP-VUE on AIX, Solaris, SUN, DEC OSF/1 and AT&T GIS operating systems.

A customer survey was conducted by HP in June 1994 to help determine the user categories most concerned with migration. 213 customers were randomly surveyed. We had a 34% (73 responses) response rate. Fig. 1 shows the percentage of respondents falling into the various user categories. Respondents were allowed to select multiple categories. For example, almost all of the respondents were end users, and 65% of them were system administrators.

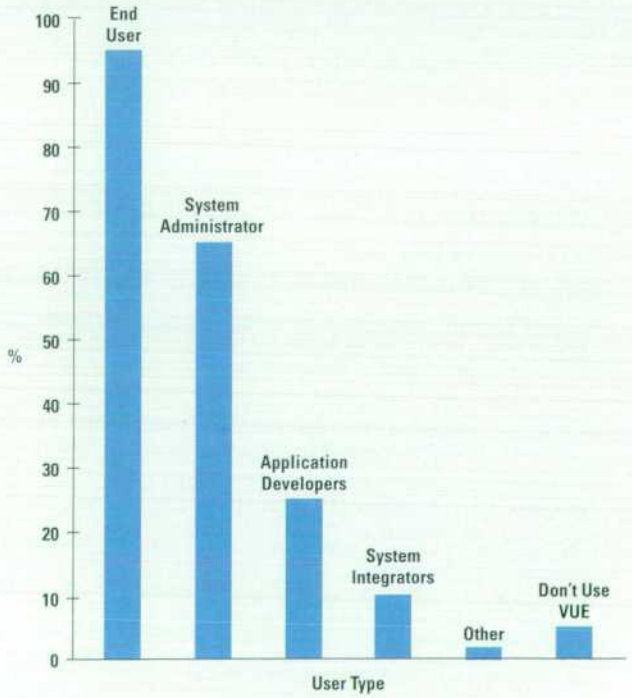


Fig. 1. The categories of users interested in migration tools.

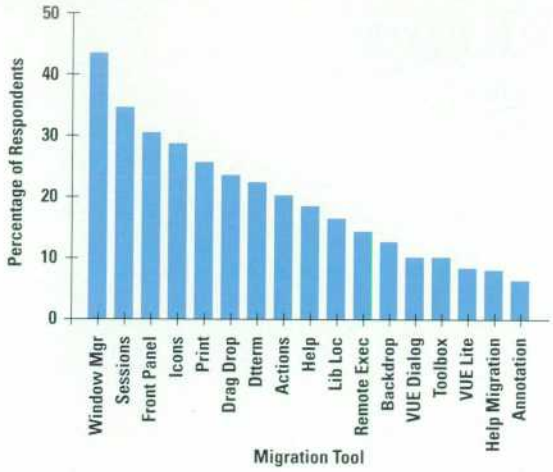


Fig. 2. The areas of interest for the type of migration tools.

The survey listed areas of HP VUE customizations and asked customers to choose the areas for which CDE must provide a migration mechanism. Fig. 2 shows this list and the percentage of respondents choosing various areas for migration.

It was decided early that the migration tools would be a set of shell scripts, modularized so that customers could run them to suit their specific needs. It was also recognized that although converters would be written to deal with the common kinds of customizations, it would be impossible to cater to all types and forms of customization. The tools would be written to be noninvasive and not change any of the HP VUE customizations.

The Difference Between HP VUE and CDE

A brief look at the major differences between HP VUE and CDE will help explain what the various converters written for migration have to accomplish. Converters were written for:

- Actions and file types
- Icon image files
- Toolboxes
- Front-panel and session customizations

Actions and File Types. Actions make it easier to run applications by representing the applications as icons that can be manipulated. When an action is created, the application is integrated into the user's graphical environment, teaching the desktop how to run the application.

A file type (referred to as a data type in CDE) provides the ability for different types of files to have different appearance and behavior. The file-type accomplishes this by:

- Defining a unique icon for the each file in the file manager windows
- Providing a custom actions menu for tasks that users do with data files
- Providing context sensitivity for actions (for example, different versions of the print action can be written for different file types, creating a file-type-sensitive printer control for the front panel).

Both HP VUE and CDE use actions and file (data) typing in the same way. Both maintain a database of actions and file type.

File types, used in conjunction with actions, can be thought of as components that create a grammar for a user's system. If files are thought of as nouns, then file types are the adjectives and actions are the verbs. Like grammar, the pieces are related to one another. There are rules that govern how they are put together and how they affect one another. It's these rules that have changed between HP VUE and CDE. This can be better illustrated with action and file type definitions in the two environments (see Fig. 3).

While action definitions have undergone only minor syntactical changes between HP VUE and CDE, the file type definitions have undergone some major changes between the two

HP VUE

```
ACTION ImageViewClient
  ARG-TYPES     *
  TYPE          COMMAND
  WINDOW-TYPE   NO-STDIO
  EXEC-HOST     hpcvusa
  EXEC-STRING   usr/bin/X11/imageview\
                   %(File)Arg_1%
  L-ICON        imagevw.l
  S-ICON        imagevw.s
  DESCRIPTION   This action invokes the\
                image Viewer, on the\
                client side if possible
  END

FILETYPE BM
  L-ICON        bitmap.l
  S-ICON        bitmap.s
  ACTIONS       ImageViewClient
  DESCRIPTION   A BM file contains data\
                in the X11 bitmap format.
  FILE-PATTERN *.bm
END
```

CDE

```
ACTION ImageViewClient
{
  ARG-TYPE      *
  TYPE          COMMAND
  WINDOW-TYPE   NO-STDIO
  EXEC_HOST     hpcvusa
  EXEC_STRING   /usr/bin/X11/imageview\
                   %(File)Arg_1%
  ICON          imagevw
  DESCRIPTION   This action invokes the\
                Image Viewer, on the client\
                side if possible
}

DATA_ATTRIBUTES BM
{
  ICON          bitmap
  ACTIONS       ImageViewClient
  DESCRIPTION   A BM file contains data in\
                the X11 bitmap format.
}
DATA_CRITERIA BM1
{
  DATA_ATTRIBUTES_NAME BM
  NAME_PATTERN *.bm
}
```

Fig. 3. Action and file type definitions in HP VUE and CDE.

environments. In addition, the file naming convention and the locations of files that contain actions and file types have changed between HP VUE and CDE.

Unlike the file type definition in HP VUE, the CDE definition consists of the following two separate database record definitions:

- DATA_ATTRIBUTE. This definition describes the data type's name and the appearance and behavior of files of this type.
- DATA_CRITERIA. This definition describes the typing criteria. Each criteria definition specifies the DATA_ATTRIBUTE definition to which the criteria apply.

There must be at least one DATA_CRITERIA definition for each DATA_ATTRIBUTE definition. A DATA_ATTRIBUTE definition can have multiple DATA_CRITERIA definitions associated with it. DATA_CRITERIA and DATA_ATTRIBUTE records are described in the article on page 24.

To allow applications integrated into HP VUE to be integrated into CDE, a converter had to be written to do the necessary syntactical conversions. These include the syntactical differences illustrated in Fig. 3, as well as others not shown in Fig. 3. The tool also writes these changes to files with the appropriate CDE suffix.

Icon Image Files

Icons are an important part of the visual appearance of both HP VUE and CDE. Icons are associated with actions, data types, front-panel controls, and minimized application windows. For HP VUE components, icon image file names use the convention: basename.size.format, where basename is the name used to reference the image, size indicates the size of the file (l for large, m for medium, or s for small), and format indicates the type of image file (pm for X pixmaps or bm for X bitmaps). The different desktop components such as the file manager and the workspace manager choose the icon to be displayed based on the size field in the icon name.

In CDE, the same format applies. However, there are four different sizes: large (l), medium (m), small (s), and tiny(t), and the sizes are used differently. For example, by default the file manager in HP VUE uses large (.l) icons whereas in CDE the file manager expects medium icons (.m). To migrate these customized icons to the CDE desktop, a tool was written to do the necessary icon name mapping to allow the customized icons to be displayed by the different CDE desktop components.

Toolboxes

In HP VUE, toolboxes are containers for applications and utilities. Each application or utility is represented by an icon called an action icon. The toolboxes in HP VUE include:

- Personal toolbox. This toolbox is one that is personally configured with actions created by the user or copied from other toolboxes.
- General toolbox. This toolbox contains applications and utilities built into HP VUE or provided by the system administrator.
- Network toolbox. This toolbox allows the user to have access to actions on other systems.

CDE replaces toolboxes with the application manager. The application manager integrates local and remote applications into the desktop and presents them to the user in a single container. A goal of the migration tools was to incorporate the personal and general toolboxes into the CDE desktop so that the users could continue to use their favorite application or utility from the CDE desktop. A decision was made not to migrate the network toolbox because the HP VUE and CDE approaches to the application server are radically different. It is easier to configure a CDE application server than an HP VUE application server.

Workspace Manager Customizations

The workspace manager controls how windows look, behave, and respond to input from the mouse and keyboard. In HP VUE the workspace manager gets information about the front panel, window menus, workspace menus, button bindings, and key bindings from a resource file called sys.vuewmrc. In CDE this information is divided into two files, which reside in different locations: dtwm.fp (front-panel definitions) and dtwmrc (menus, button and key bindings).

Front-Panel Customizations. The front panel is a special desktop window that contains a set of controls for doing common tasks. The front panel has undergone a visual as well as a definition change between HP VUE and CDE. In the case of HP VUE, there are three main elements to the front panel: the main panel, which includes both the top and bottom row, and subpanels (Fig. 4a). In CDE there is no bottom row. Instead, some of the controls that resided in the HP VUE bottom row, such as the busy signal, exit, and lock buttons have moved to the workspace switch in CDE, while the other controls are available through the personal applications slideup (see Fig. 4b). The toolbox in HP VUE has been replaced by the application manager.

Besides the visual differences, the front-panel definitions for panel, box, and control have also changed. To state it simply, in HP VUE, a container (panel) has knowledge of its contents (box), but a box has no knowledge of its parent (panel). In CDE the reverse is true (e.g., a box knows about its parent (panel), and a control knows about its parent (box) but not vice versa. Fig. 5 shows these differences.

Since there were both syntactical and logic differences between the HP VUE and CDE front panels, this was one of the most difficult converters to write. While preserving user customizations, the converter also had to replace HP VUE controls with their equivalent CDE counterparts (e.g., replacing the toolbox with the application manager). In the case of a one-to-one mapping between an HP VUE control and a CDE control such as the mail icon, the converter has to migrate any customizations made to these controls in HP VUE or replace them with the equivalent CDE defaults.

Session Customizations. A session refers to the state of the desktop between the time the user logs in and the time the user logs out. A session is characterized by:

- The applications that are running
- The icons on the desktop
- The look (colors, fonts, size, location, etc.) of application windows
- Other settings controlled by the X server such as mouse behavior, audio, volume, and keyboard click.

Sessions are divided into two categories: current sessions and home sessions. A current session is one that is stored at logout so that when a user logs in again the session state at

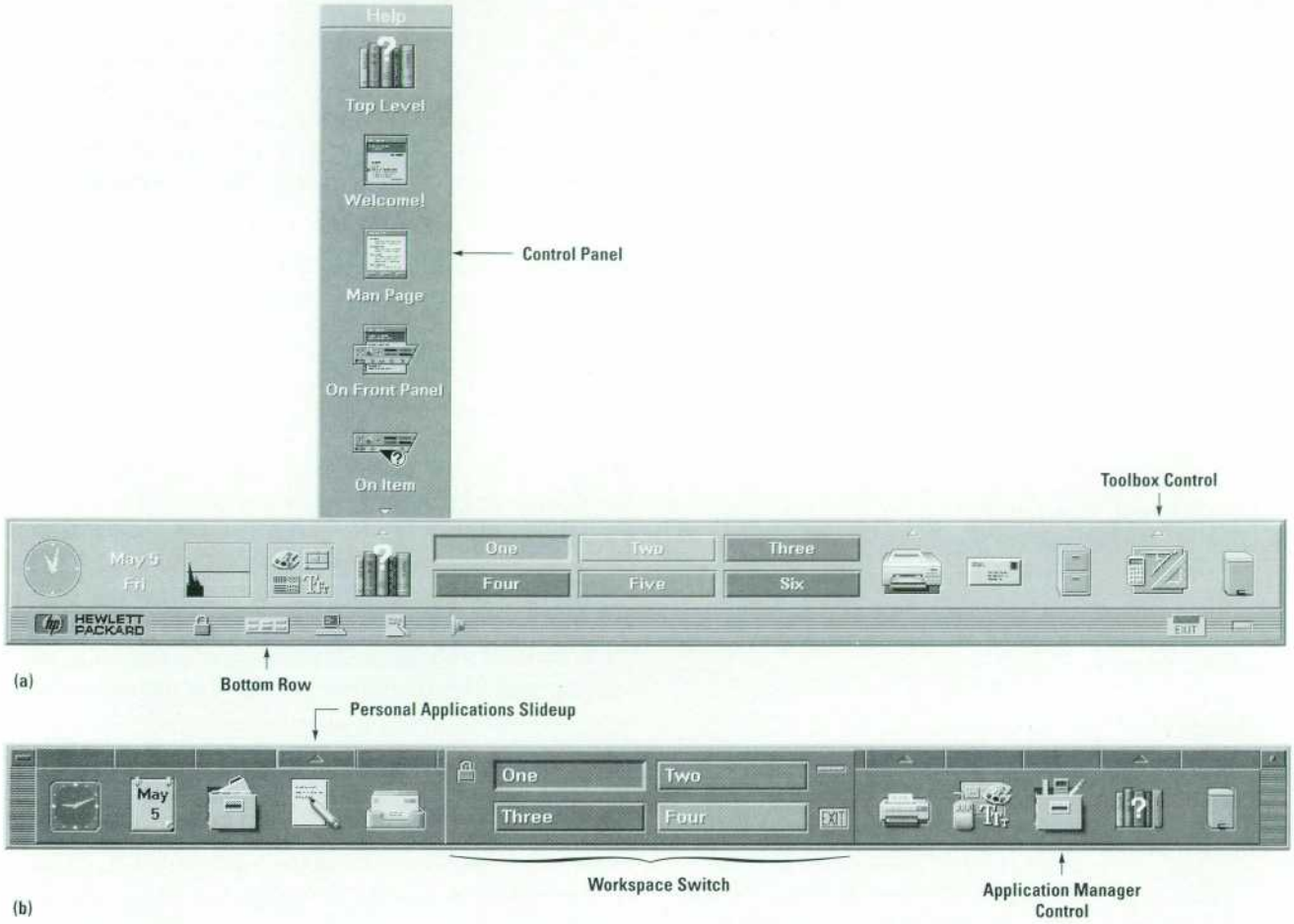


Fig. 4. (a) HP VUE default front panel. (b) CDE default front panel.

logout is reestablished. A home session, which is explicitly stored by the user at some other time during a session, allows the user to return to some known session.

Session information is stored in the files listed in Table I.

Table I
Session Files

File	Contents
vue.session	The names of active clients and their window geometries, workspaces, states (normalized or minimized), and startup strings.
vue.resources	The resources for the active clients (including the workspace manager) in the session.
vue.settings	Server and session manager settings such as screen-saver timeout, audio, and keyboard repeat.

Since the goal of the migration tools was to allow a seamless transition from HP VUE to CDE, we determined that it was necessary to bring over the users' sessions, so that their CDE sessions resembled HP VUE sessions as much as possible. A converter was written to convert the appropriate session files to their CDE equivalents. This meant creating equivalent dt.session, dt.resources, and dt.settings files with the names of HP VUE clients being replaced with their CDE counterparts if they existed. For example, substituting Vuewm with Dtwm and Vuefile with Dtfile.

A Graphical User Interface for Migration Tools

While the individual converters were being developed, a graphical user interface for the converters was being designed. Migration tool users can be classified into two groups: system administrators and end users. The interfaces for the migration tool had to be tailored to the needs of these two types of users even though the converters being used were essentially the same. This involved several iterations on paper of the various dialogs that would take the user through the migration process. Since all of the converters were written using shell scripts to allow easier modification by the customer if needed, a decision was made to use dtksh for developing the GUI. Dtksh is a utility that provides access to a powerful user interface from Korn shell scripts. Dtksh is part of the CDE API.

The different converters were gathered together into a migration application called VUEtoCDE. To run the migration tools, the VUEtoCDE migration product has to be installed on the system after CDE has been installed. The first time the user (system administrator or end user) logs into the system the migration dialog box (Fig. 6) appears if the user's home directory contains a .vue directory. The assumption here is that the user has used HP VUE and might choose to move

HP VUE

```
PANEL FrontPanel          /* Parent references
                                    child */
{
 BOX              Top     /* child */
 BOX              Bottom  /* child */
}


BOX Top
{
 TYPE             primary
 CONTROL          Clock   /* child */
 CONTROL          Date    /* child */
 CONTROL          Load    /* child */
 •
 •
 •
}
CONTROL Clock
{
 TYPE             clock
 SUBPANEL         ClockSubpanel
 HELP_TOPIC       FPClock
}


BOX ClockSubpanel
{
 TYPE             subpanel
 TITLE            "Time Zones"
 HELP_STRING      "This subpanel contains
                  time-related funtions"
 CONTROL          JapaneseClock
}

CONTROL JapaneseClock
{
 TYPE             button
 IMAGE            clock
 PUSH_ACTION      f.action TimeJapanese
 LABEL            "Japan"
}
```

CDE

```
PANEL FrontPanel          /* Parent has no references to
{                                    children */
 DISPLAY_HANDLES          True
 DISPLAY_MENU             True
 DISPLAY_MINIMIZE         True
 CONTROL_BEHAVIOR         single_click
 DISPLAY_CONTROL_LABELS   False
 HELP_TOPIC               FPOnItemFrontPanel
 HELP_VOLUME              FPanel
}

BOX Top
{
 CONTAINER_NAME FrontPanel  /*child references parent */
 POSITION_HINTS first
 HELP_TOPIC     FPOnItemBox
 HELP_VOLUME    FPanel
}



CONTROL Clock
{
 TYPE           clock
 CONTAINER_NAME Top          /* child references parent */
 CONTAINER_TYPE BOX
 POSITION_HINTS 1
 ICON           Fpclock
 LABEL          Clock
 HELP_TOPIC     FPOnItemClock
 HELP_VOLUME    FPanel
}



SUBPANEL ClockSubpanel
{
 CONTAINER_NAME CLOCK        /* child references parent */
 TITLE          "Time Zones"
}
CONTROL JapaneseClock
{
 TYPE           icon
 CONTAINER_NAME ClockSubpanel/* child references parent */
 CONTAINER_TYPE SUBPANEL
 POSITION_HINTS 1
 ICON           clock
 LABEL          Japan
 PUSH_ACTION    TimeJapanese
 HELP_TOPIC     FPOnItemClock
 HELP_VOLUME    FPanel
}
```

Fig. 5. Front panel definitions in HP VUE and CDE.

HP VUE 3.0 customizations to CDE. Several options are provided by which the user could migrate now or later or not at all. The user can invoke the tool from the application manager at any time.

If the user chooses to migrate, the tool determines if the user is a system administrator or an end user and presents the appropriate dialog. Fig. 7 is the dialog presented to the end user when the Yes,Now button is selected.

The thinking here was that more often than not, the user would choose the one-step migration (default option), but we wanted to provide choices for those who didn't want the default option. Also, by choosing the various converter options, the user can exercise a specific converter in the event that it did not work as expected in the one-step migration.

Fig. 6. Initial migration dialog.

The converters were designed to avoid modifying any of the HP VUE directories, allowing the user to rerun the converters as many times as needed. In the case of actions, file types, and icons the converter expects an HP VUE source directory and a CDE destination directory. This allows the user to enter nonstandard source and destination locations for action, file type, and icon image files. When a nonstandard destination is entered, the tool displays a message that this path has to be added to the user's .dtprofile file with a specific input environment variable to update the global search path. This option was chosen over having the tool modify the user's .dtprofile, which was considered to be too invasive. Initially, the user is presented with the default HP VUE and CDE locations (see Fig. 8), but the user can type in as many source and destination directories as desired.

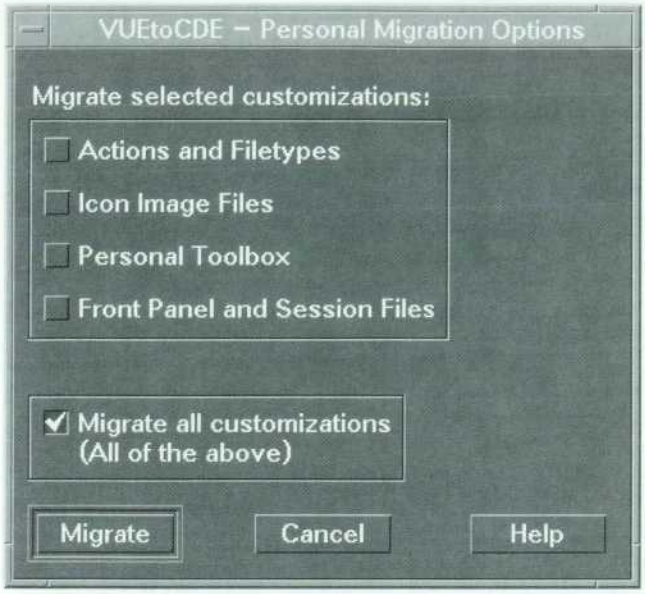


Fig. 7. User-level migration.

Because of the shift from toolboxes in HP VUE to the application manager in CDE, the migration of a toolbox to any location other than the CDE application manager is not straightforward. It requires some in-depth understanding of how to integrate applications into the CDE desktop before the user can specify a nonstandard destination for an HP VUE toolbox. Thus, the migration tool disallows specifying a nonstandard destination and only provides the flexibility of choosing the name of the application group for an HP VUE toolbox (see Fig. 9).

Each converter that is run generates a log file in a directory prepended with the user's login name (e.g., mollyj_Vue2Cde_Log) in /tmp, allowing for faster diagnosis of problems.

When the user has completed the personal migration, a final dialog is displayed that contains a special button for exiting the current CDE session. This button is necessary because of the design of the CDE session manager. Ordinarily, when the user logs out, the contents of the current session are written to the current session configuration files, overwriting the previous contents. Thus, if the user were to simply log out after migration, the migrated session files would be overwritten. To prevent this problem, the CDE session manager provides a special termination option used only by the VUEtoCDE tool. This option ends the current session without saving it so that the migrated session information is preserved and available at next login.

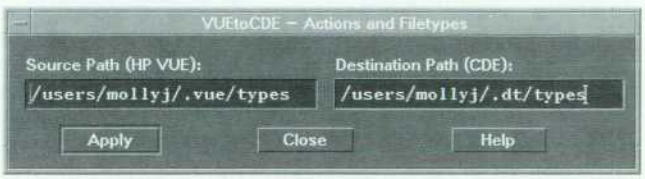


Fig. 8. Actions and file types conversion.

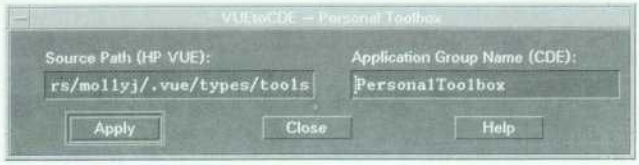


Fig. 9. Toolbox conversion.

Migration of a User's Front-Panel Customizations

Consider the simple case of an HP VUE front panel that has been customized to add a media control with a corresponding subpanel in the top row and an icon box in the bottom box (see Fig. 10). The user running the migration tools would expect these controls to be migrated over to the CDE front panel. The results are shown in Fig. 11. The media subpanel has been migrated to the main CDE front panel and the icon box to the personal applications subpanel because of the absence of a bottom row in the default CDE front panel. The personal applications subpanel and the application manager icon are placed after the trash icon. These two controls do not have analogs in HP VUE, and a decision had to be made regarding the placement of these two controls on the converted front panel. Since an HP VUE front panel could take any shape or form (multiple rows versus just a top and bottom row), a definitive location was needed for these two controls. A decision was made to place them as the last two controls of the last row in the converted front panel. This conversion can be achieved either by clicking the Migrate button on the dialog box shown in Fig. 7, which results in migrating all customizations, or by selecting from the same dialog box one of the following options and then clicking the Migrate button:

- Actions and File types
- Icon Image Files
- Front Panel and Session Files

Now that we have illustrated how the tools convert a front panel with some relatively simple customizations, we would like to illustrate what the migration tool can do in cases where the customizations are complex and unconventional.

Fig. 12 shows an example of a highly customized HP VUE front panel. It's a box with five rows and several subpanels. During the development phase of our project this panel was considered to be the ultimate test for the front-panel converter. This front panel helped shatter several assumptions made in the design and pushed the limits of the front-panel converter. The desire here was to preserve the row positions and subpanels while modifying the front panel so that it conformed to the CDE default front panel. What we ended up with is a reasonably accurate conversion (Fig. 13). The HP VUE front panel had five rows, one of which was the bottom row. This translated to four rows in CDE since the default CDE front panel does not have a bottom row. The HP VUE controls in the bottom row (e.g., lock, exit, and busy controls), for which there are CDE equivalents, are not migrated unless they have been customized. This also applies to the terminal window and text editor controls in the bottom row

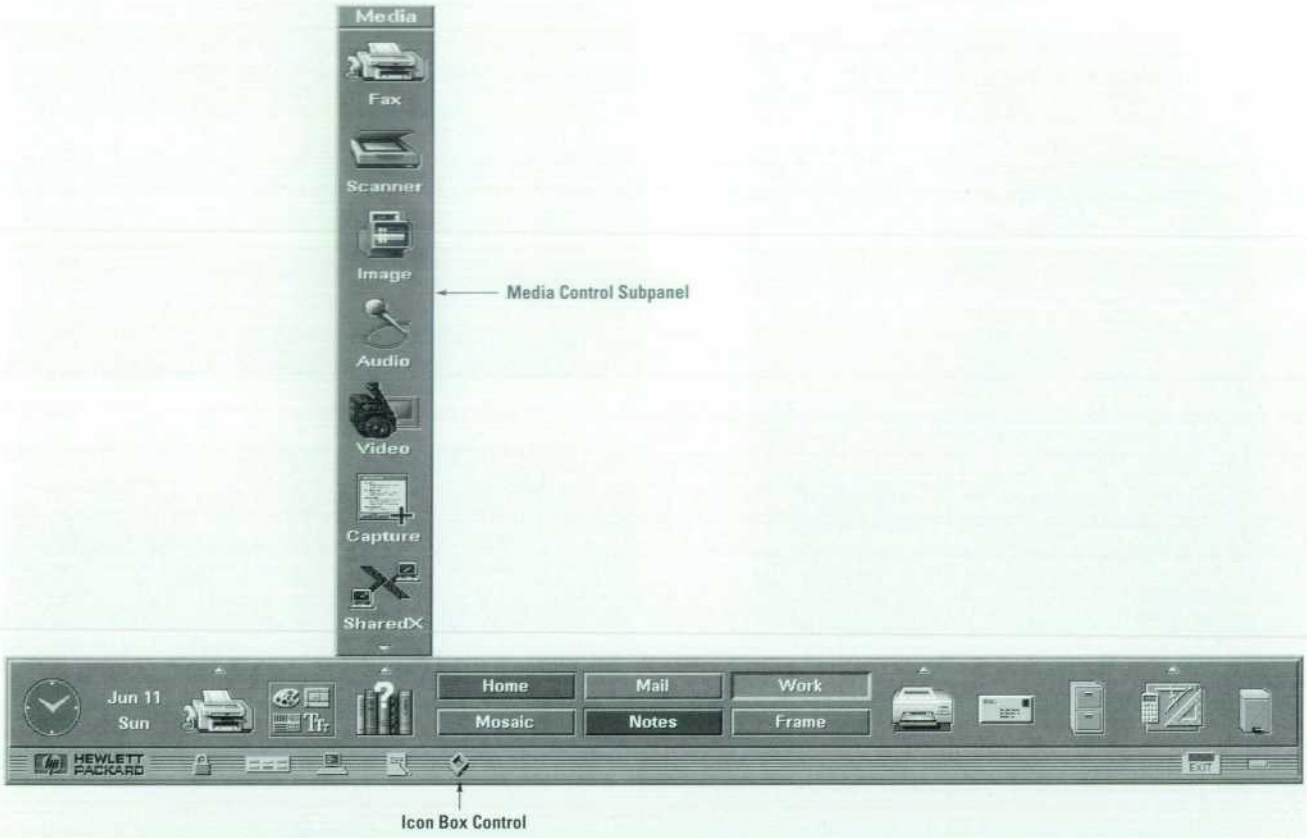


Fig. 10. User's customized HP VUE front panel.

for which there are CDE equivalents in the personal applications slideup. Any other customized controls in the bottom row are also moved to the personal applications slideup.

HP VUE controls that do not map to controls in CDE are removed. Keeping this in mind, note the one-to-one mapping of controls to rows between the HP VUE panel in Fig. 12 and the CDE panel in Fig. 13. What is unexpected is the extra space seen at the end of rows in the converted CDE front panel. The reason for this is that the application manager and the personal applications slideup, which have no analogs in HP VUE, need a definitive location in the converted front panel. The decision to place them as the last two controls in the last row of the converted front panel has resulted in this row being rather long, creating the extra space. Other factors that have contributed to this extra space are the area around the workspace switch, which did not exist in HP VUE, the space taken by the window frame, which was missing in the HP VUE front panel, and the size of the default workspace buttons in CDE, which are wider than their HP VUE counterparts.

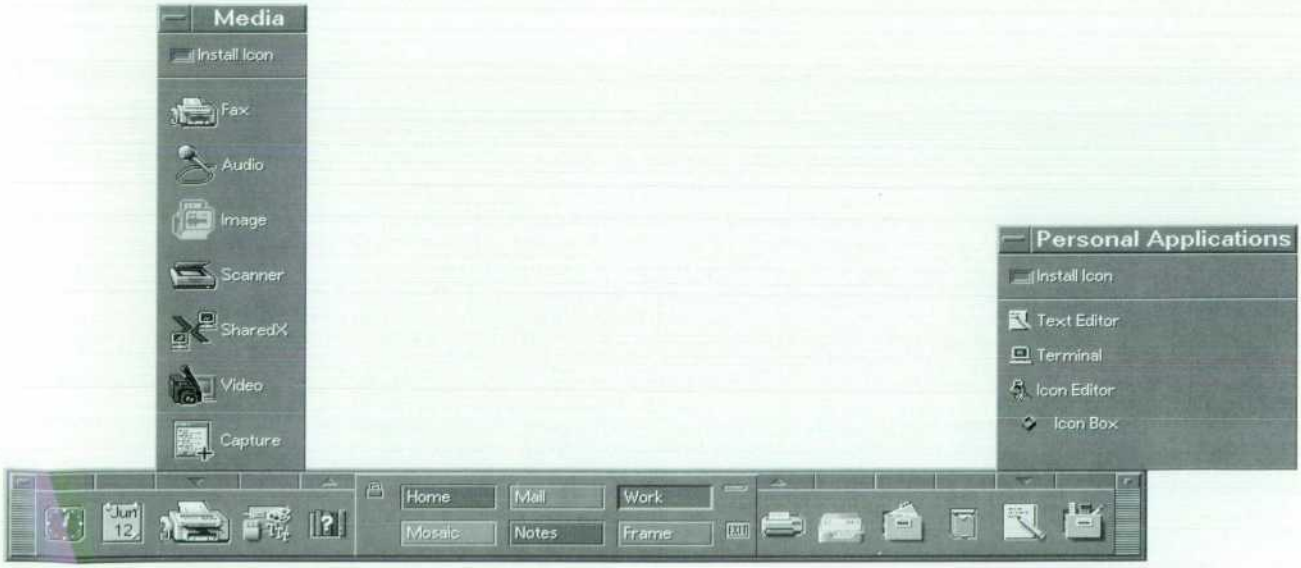


Fig. 11. Converted CDE front panel.

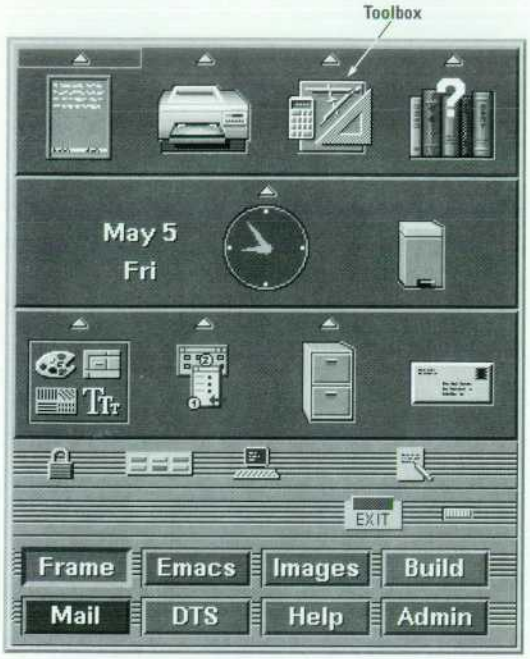


Fig. 12. A highly customized HP VUE front panel.

System-Level Migration

In designing the system-level migration, the thinking was that system administrators typically do not like one-step migrations but would much rather do a step-by-step migration to allow more control. Exactly the same converters available to the user are available here with the defaults being set to system level. Log files for the converters are created in a directory called System_VuetoCde_Log in /tmp. The system-level migration options and the front-panel and session files conversion menu are shown in Fig. 14.

Remote Execution from the CDE Desktop

In addition to the options specified in the VUEtoCDE-System Migration Options dialog in Fig. 14a, a command-line converter is available to create a minimum CDE fileset for remote execution.

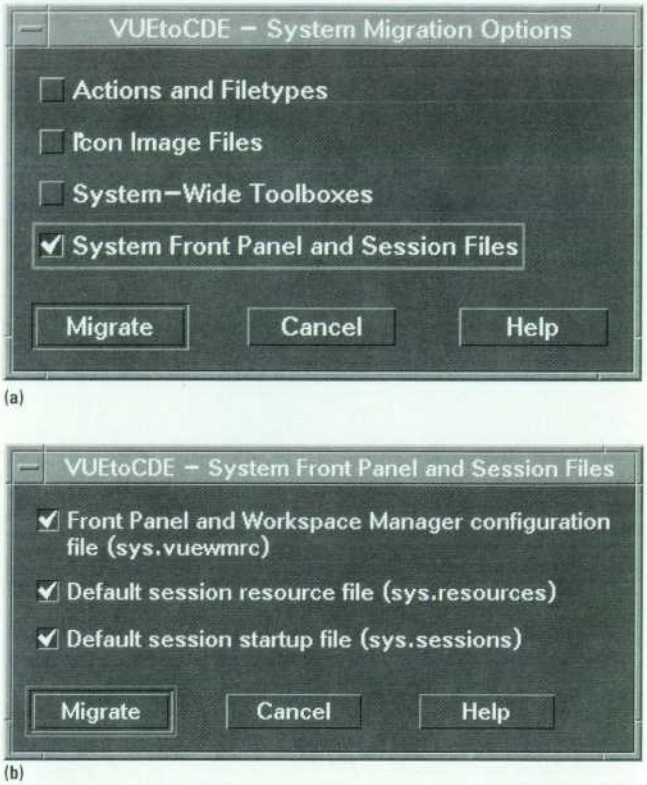


Fig. 14. (a) System-level migration options. (b) Front-panel and session files conversion.

The fileset required on a remote machine for an HP VUE action to execute remotely is different from that required for CDE remote execution. While HP VUE requires the spcd service, CDE relies on dtspcd and ToolTalk® services. Spcd, dtspcd, and ToolTalk are services that allow actions to invoke applications that run on remote machines. A converter was written to build the minimum CDE fileset that could then be installed on the remote machine (if CDE is not already installed) to allow actions to continue working from the CDE desktop after migration. This converter does not provide as complete a migration as some of the other converters

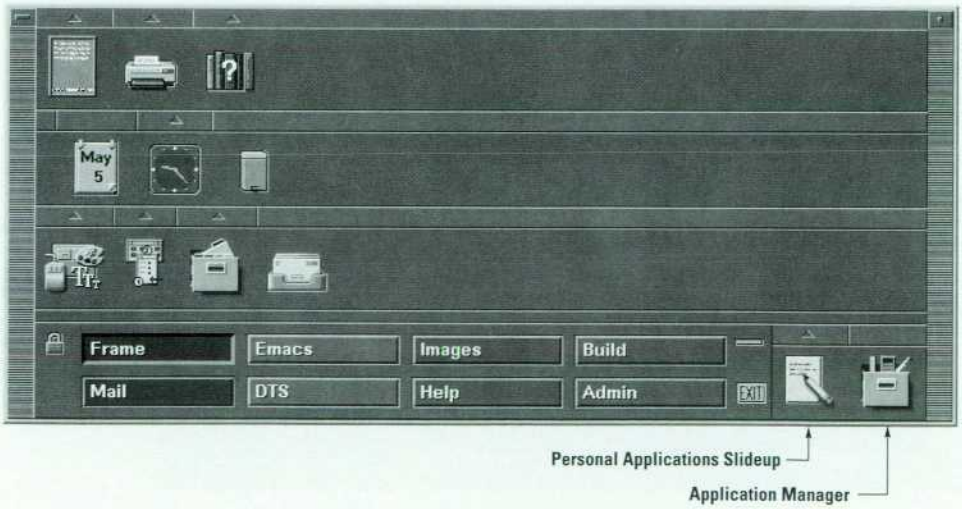


Fig. 13. A migrated CDE front panel.

because system files have to be modified manually as well. Nevertheless it was felt that any form of help provided during this transition from HP VUE to CDE would be worthwhile.

Usability Testing

The migration tools underwent usability testing at HP's Corvallis and Fort Collins laboratories. Several system administrators and end users were asked to run the migration tools. These sessions were remotely observed by the engineer conducting the tests. These sessions were also videotaped and reports were put together by the usability engineer and sent to the learning products and development engineers. This input was used to further modify the tools, online help, and documentation. Most often the confusion was over the choice of names for buttons or not having enough feedback when a task was completed.

Conclusion

Although the migration tools described here do not provide a complete migration from HP VUE to CDE, the converters do bring over a large portion of a user's customizations through an easy-to-use graphical user interface. Since HP is committed to its customers, we will continue to support HP VUE as long as our customers want it, but we hope that these tools will be an incentive for HP VUE users to embrace CDE sooner rather than later and make CDE the true desktop standard.

Acknowledgments

I would like to acknowledge Steve Beal, from TriTeal Corporation, who collaborated in developing the converters for the HP VUE to CDE migration suite. Special thanks to Anna Ellendman, Julia Blackmore, and Ellen McDonald of the learning products team for their valuable user interface design reviews and testing. I also wish to acknowledge Jay Lundell, a former HP employee, for his insightful human factors review and project managers Ione Crandell and Bob Miller for their support.

A Media-Rich Online Help System

Based on an existing fast and easy-to-use online help system, the CDE help system extends this baseline to provide features that will work across all UNIX® platforms.

by Lori A. Cook, Steven P. Hiebert, and Michael R. Wilson

With the growing demand by users for a unified desktop strategy across all UNIX operating system platforms comes the requirement for a standard help system. Users expect some base level of online help to be provided from within the desktop they are using. They expect online information to be easy to use and graphical, with growing expectations of direct audio and video support and interactive capabilities.

The Common Desktop Environment (CDE) help system provides media-rich, online information that is both fast and easy to use. It has its basis in the standard online help system from HP that is used extensively by HP VUE 3.0 and HP MPower[1] components and by many other HP OSF/Motif-based products.

Background

The CDE 1.0 help system originated with HP VUE. An early version, HP VUEhelp 2.0, satisfied few of the requirements of a modern help system. VUEhelp 2.0 did not allow rich text and graphics, was hard to integrate into an application, lacked authoring tools, and suffered from poor performance. The HP VUE 3.0 help system delivered a complete solution for creating, integrating, and shipping rich online information with an OSF/Motif-based application, while keeping its presence (use of system resources) to a minimum. HP VUE 3.0 walked a very fine line between providing the rich set of features that customers required while maintaining performance. In 95% of the cases where features and performance came into conflict, performance won. The HP VUE 3.0 help system was submitted to the CDE 1.0 partners for consideration. It was the only existing, unencumbered, rich text help system put forward for consideration. After an evaluation period, the HP VUE 3.0 help system was accepted almost as is. It contained all the functionality required, and with a little work, it could use a standard distribution format.

The Help Developer's Kit

The CDE 1.0 developer's kit includes a complete system for developing online help for any OSF/Motif-based application. It allows authors to write online help that includes rich graphics and text formatting, hyperlinks, and communication with the application. It provides a programmer's toolkit that allows developers to integrate this rich online help information with their client applications. The help dialog widget serves as the main display window for the CDE 1.0 help system. A second, lighter-weight help widget, called *quick help dialog*, is also available from the toolkit. Following is the list of components supported within the toolkit:

For authors:

- The CDE 1.0 HelpTag markup language. This is a set of tags used in text files to mark the organization and content of online help. HelpTag is based on SGML (Standard Generalized Markup Language).
- The CDE 1.0 HelpTag software. This is a set of software tools for converting the authored HelpTag files into their runtime equivalents used to display the online help information. CDE 1.0 HelpTag is a superset of the HelpTag used with HP VUE 3.0. HP VUE 3.0 HelpTag source files compile under CDE 1.0 with no modification. One exception to this is if authors use the old HP VUE 3.0 help distribution format.
- The dthelpview application. This program is for displaying online help so it can be read and interacted with in the same manner as end users will use it.

For programmers:

- The DtHelp programming library. This is an application programming interface (API) for integrating help windows (custom OSF/Motif widgets) into an application.
- A demonstration program. This is a simple example that shows how to integrate the CDE 1.0 help system into an OSF/Motif application.

Changed or new features for CDE 1.0:

- Public distribution format based on SGML conforms to standards.
- A new keyword searching capability allows users to search across all volumes on the system.
- A new history dialog allows the user to select previously viewed volumes.
- A different table of contents mechanism allows full browsing of the volume without using hypertext links.
- A richer message formatting capability is provided. (While HelpTag does not allow tables, other documents that do allow tables can be translated into the public distribution format and displayed. Also, the table of contents now allows graphics in the titles.)
- Public text extraction functions are removed, but available for old HP VUE 3.0 users by redefining.

General Packaging Architecture

An online help system needs to feel like it is part of the host application to the end user, not an appendage hanging off to the side. For developers to leverage a third-party help system, it must be delivered in such a way as to provide easy and seamless integration into their applications. Furthermore, the effort and overhead of integrating and redistributing the help system along with their applications must be minimal,

while at the same time meeting application and end-user requirements for a help system. Users should feel as if they have never left the application while getting help.

During the initial prototyping of the HP VUE 3.0 help system, two key issues kept occurring: performance and packaging. The help system needed to be light and fast and easy to integrate into any OSF/Motif-based application and redistribute with that application. HP VUE 2.0 help suffered greatly on both these points. VUEhelp 2.0 was server-based, large, slow, and dependent on the HP VUE desktop. Any application using this help service had to run within the HP VUE desktop environment.

These two issues were addressed in HP VUE 3.0 help and we found no reason to change the paradigm in the CDE development. To fix the performance problems (slow startup times), functionality is linked into the client via a library rather than starting up a separate help server process (as was the case for HP VUE 2.0). Since most OSF/Motif applications tend to incorporate the same widgets as those used by the help dialogs, the increase in data and text for attaching a help dialog to an application is minimal.

Fig. 1 represents two different approaches to integrating online help into an application. Our initial prototypes quickly exposed the value of a client-side embedded solution especially with respect to performance (e.g., fast rendering time) and memory use.

The following advantages are gained by using a library-based help system instead of a server-based architecture.

Integration advantages:

- OSF/Motif-based API (simple to use for Motif knowledgeable developers)
- Application control of the help system dialog management (creation, destruction, caching, and reuse)
- Smooth transition into help system via consistent resource settings between the application and the help system (same fonts and color scheme and quick response times)
- Ready support for a tightly coupled application-to-help-system environment (application defined and processed help controls such as hypertext links)
- Application-specific customizing of the help system (dialogs, controls, resources, and localization).

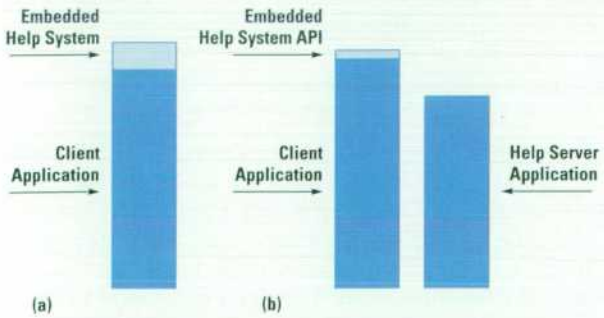


Fig. 1. Two different approaches to integrating online help into an application. (a) Client-side embedded implementation. This is the scheme used in CDE 1.0. (b) Server-side implementation of the help system. This was the implementation used in HP VUE 2.0.

CDE dependencies:

- DtHelp is delivered in shared-library form only.
- Application must link with the CDE libraries DtSvc and tt.
- CDE is not required as long as the libraries it depends on are available.

Integration Concepts, Practices, and Mechanisms

The intricacies of how to author online information, or the different ways a developer can integrate help into an application, are described in the CDE 1.0 help system developer's guide. We will describe in this section the general integration concepts and practices for which this help system was intended.

The run-time help facility is made up of a collection of help dialogs (complex, composite widgets), and compiled help volumes (described below). The help widgets are linked directly into the client application via the help library libDtHelp.sl (shared library) and instantiated by the client to display help information. The help dialogs serve only as the vehicle for displaying rich online help information, while standard OSF/Motif, Xlib, and Xt services serve as the glue to integrate them into the application. For this reason, it is necessary to stray from discussing the specifics of the help system and its components, and describe some of the OSF/Motif and toolkit mechanisms that must be used to integrate help into an application.

There are two levels of integration with respect to this help system: integrating help into an application and integrating a help-smart application into CDE.

Integrating Help into an OSF/Motif Application

Developers have many degrees of freedom with respect to how much or how little help they include in their applications. If an application and its help information have very loose ties, there may be only a handful of topics that the application is able to display directly. In contrast, the application could provide specific help for nearly every object and task in the application. This requires more work, but it provides potentially much greater benefits to the end user if done well.

Help menus and buttons in an application serve as the most basic of help entry points for an application. Reference 2 provides more information about integrating help menus and buttons into an application.

Contextual Help. Contextual help provides help information about the item on which the selection cursor* is positioned. Contextual help information provides users with information about a specific item as it is currently used. The information provided is specific to the meaning of the item in its current state.

The OSF/Motif user interface toolkit provides direct support for contextual help entry points via its help callback mechanism. When a valid help callback is added to a widget and the user presses the help key (**F1**) while that widget has the current keyboard focus (selection cursor), the widget's help callback is automatically executed. From within the help

* A selection cursor is visual cue that allows users to indicate with the keyboard the item with which they wish to interact. It is typically represented by highlighting the choice with an outline box.

callback the application has the opportunity to display some help topic directly based on the selected widget, or the application could dynamically construct some help information based on the current context of the selected item.

Any level of granularity can be applied when adding help callbacks to an application's user interface components. They can be added to all the widgets and gadgets within the application dialogs, to the top-level windows for each of the dialogs, or to any combination in between.

If the user selects **F1** help with the selection cursor over a widget or gadget that has no help callback attached to it, the OSF/Motif help callback mechanism provides a clever fallback mechanism for providing more general help. The help callback mechanism will jump to the nearest widget ancestor that has a help callback assigned, and invoke that callback. The theory is that if you don't have a specific help on that widget or gadget, then it's better to provide more general help than none at all.

Application developers are responsible for adding their own help callbacks to their user interface components within their application. OSF/Motif sets the help callbacks to NULL by default.

Item Help. Item help allows users to get help on a particular control, such as a button, menu, or window, by selecting it with the pointer. Item help information should describe the purpose of the item for which help is requested and tell users how to interact with that item. This information is typically reference-oriented.

Item help is usually accessed via an application's help under the On Item menu selection. Once selected, the selection cursor is replaced with a question mark cursor and users can choose the item on which they want help by making the selection over that item.

The CDE help system API utility function DtHelpReturnSelectedWidgetId() assists developers in providing item help within their applications. This function provides an interface for selection of a component within an application.

DtHelpReturnSelectedWidgetId() will return the widget ID for any widget in the user interface that the user has selected via the pointer. The application then has the opportunity to display some help information directly on the selected widget or gadget.

At any point while the question mark cursor is displayed, the user can select the escape key (**ESC**) to abort the function call. If the user selects any item outside the current applications window, the proper error value will be returned.

Once DtHelpReturnSelectedWidgetId() has returned the selected widget, the application can invoke the help callback on the returned widget or gadget to process the selected item.

From within the help callback, the application has the opportunity to display some help topics based on the selected widget, or it could dynamically construct some help information based on the current item selected.

Integrating a Help-Smart Application into the Desktop

There are no restrictions regarding where run-time help files are installed. However, a suggested practice is to set up the help file installation package with the ability to redirect the default help file install location (e.g., put them on a remote help server system). By default, the run-time help files should be installed with the rest of an application's files. Installing the run-time help files in the same location as the application's files gives system administrators more flexibility in managing system resources.

An important step in installing help files is registration. The registration process enables two important features of the CDE 1.0 help system: cross volume hyperlinks and product family browsing.

Registering Help Volumes. After the run-time files have been installed, the volume is registered by running the CDE 1.0 utility, dtappintegrate. This utility registers the application's help volume by creating a symbolic link from where the help volumes are installed to the registry directory /etc/dt/appconfig/help/<$LANG>. By using this utility to register the help volumes, the application can ask for a help volume by its name only and the DtHelp library will search the standard locations for the volume's registry. Then, no matter where the application's help volume ends up, the DtHelp library finds the symbolic link in a standard location and traces the link back to the real location. The article on page 15 describes registration and the utility dtappintegrate.

If access to an application's help volume is restricted to the application, then the location of the help volume can be hard-coded into the application. The disadvantage of this method occurs when a system administrator moves the application's help volumes to another location. If the application looks in only one place for the help information, moving the help volume causes a serious problem.

Registering a Product Family. When registering a product family, which is a group of help volumes belonging to a particular product, a help family file (product.hf) should be created and installed with the rest of the product's help files. The dtappintegrate utility creates a symbolic link for the product file to a standard location where the browser generator utility, dthelpgen, can find and process it. The dthelpgen utility creates a special help volume that lists the product families and the volumes within each family installed on the system.

Access to Help Volumes

The CDE 1.0 help system has a simple, yet extensible mechanism for transparent access of help volumes installed on the desktop. It supports both local and remote access to help volumes and works with any number of workstations. The only dependencies required are proper help registration (discussed above), NFS services (i.e., remote systems mounted to the local server), and proper configuration of the help environment variables discussed below.

When an application creates an instance of a help widget the DtNhelpVolume resource can be set using one of two formats: a complete path to the volume.sdl file, or if the volume is registered, the base name of the volume. When using the base name, the help system searches several directories for the volume. The search ends when the first matching volume is found. The value of the user's $LANG environment variable is also used to locate help in the proper language (if it's available).

DTHELPUSERSEARCHPATH. This environment variable contains the user-defined search path for locating help volumes. The

default value used when this environment variable is not set is:

- dt/help/%T/%L/%H: \
- dt/help/%T/%L/%H.sdl: \
- dt/help/%T/%L/%H.hv:

where:

- %L is the value of the $LANG environment variable (C is the default)
- %H is the DtNhelpVolume (help volume) resource specified
- %T is the type of file (volume or family) being searched for.

Whenever this resource is set to a relative path, the resource value will be prefixed with the user's default home directory.

Examples:

Help volume: /user/vhelp/volumes/C/reboot.hv
Product family: /user/vhelp/family/SharedX.hf.

The .hv and .hf extensions are provided for backwards compatibility with the HP VUE 3.0 help and family volumes.

DTHELPUSERSEARCHPATH supplements any system search path defined. Therefore, the user uses this environment variable to access help volumes that should not be available to everyone.

DTHELPSEARCHPATH. This environment variable specifies the system search path for locating help volumes. The default values used when this environment variable is not set include:

- /etc/dt/appconfig/help/%T/%L/%H: \
- /etc/dt/appconfig/help/%T/%L/%H.sdl: \
- /etc/dt/appconfig/help/%T/%L/%H.hv: \

Where: %L, %H, and %T are the same as above.

When setting either of these environment variables it is a sound practice to append the new values to the current settings.

Help Widgets

The CDE 1.0 help dialog widgets are true OSF/Motif widgets. The syntax and use model for programmers is the same as for every OSF/Motif widget, whether custom or part of the toolkit. This makes it easy for developers familiar with OSF/Motif programming to understand and use help widgets.

The OSF/Motif-based API directly supports various controls to manage an instance of a help dialog. Through standard OSF/Motif, Xt, and Xlib functions, programmers access help dialog resources and manipulate these values as they see fit. Developers add callbacks (XtAddCallback()), set and modify resources (XtSetValues()), manage and unmanage the dialogs (XtManageChild, and XtUnmanageChild) and free resources (XtDestroyWidget()).

The General Help Widget

The general help widget for CDE 1.0 is very similar to the general help widget in HP VUE 3.0 help. It contains a menu bar, a table of contents area, and a topic display area. Added for CDE 1.0 are buttons to the right of the table of contents for easy access to menu items. Behavior for the table of contents area and keyword searching has changed for CDE 1.0. Fig. 2 shows a general help widget.

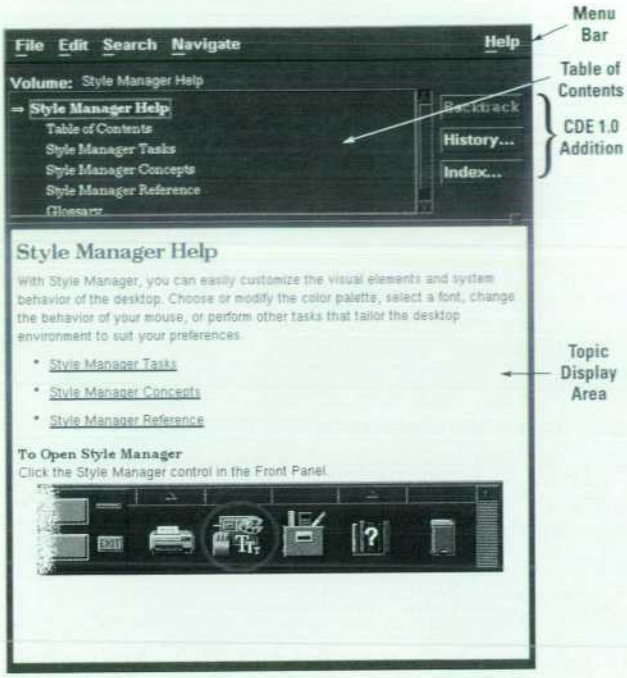


Fig. 2. General help dialog box.

New Buttons. For convenience, buttons for the backtrack functionality, history, and index dialog boxes have been added to the general help dialog widget for CDE 1.0.

Table of Contents. The table of contents area changed significantly from its HP VUE 3.0 help system predecessor. The old version was a "here you are" type of listing, and it did not give the user the chance to explore other side topics unless they were direct ancestors of the current topic. Consequently, this type of table of contents required the authors to write help topics with extensive hypertext linking to other topics such as child topics that would be of direct interest to the user or direct descendants of the current topic. Fig. 3 shows the difference between HP VUE 3.0 and CDE 1.0 table of contents boxes.

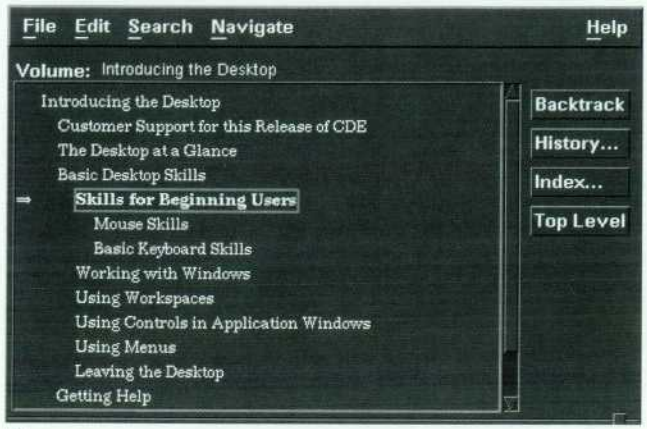
CDE 1.0 help shows the topic, the subtopics related to the current topic, the ancestor topics, and the ancestor's sibling topics. As the user selects other topics from this table of contents, the listing changes so that it always shows the path to the topic, the siblings of any topic in the path, and the children of the current topic. Thus, the writer only has to include hypertext links to topics of interest if the topic is not a child of the current topic, an ancestor topic, or an ancestor's sibling.

Another change is that the table of contents now displays graphics. Before, the table of contents was limited to text. Now, the table of contents displays multifont and graphical headings.

Searching for a Keyword. The search dialog for CDE 1.0 help has been completely redesigned. A common complaint was that the old HP VUE keyword search dialog displayed or knew about keywords for the current volume only.

Now the user has the option to search all known volumes on the machine for a keyword. Since this can be a time-intensive

(a)



(b)

Fig. 3. Help table of contents for (a) HP VUE 3.0 and (b) CDE 1.0.

operation, the search engine allows the user to interrupt a search operation. Additionally, the user can constrain the search to a limited set of volumes. Fig. 4 shows the new CDE 1.0 search dialog box.

Quick Help Dialog

From the programmer's point of view, the quick help dialog is a stripped-down version of the general help dialog. Its

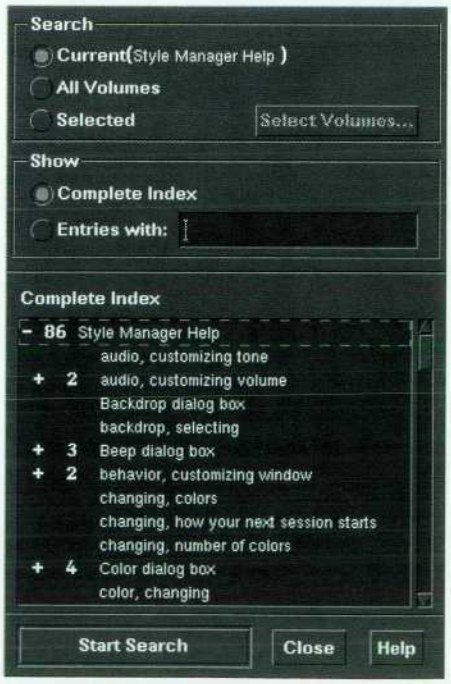


Fig. 4. CDE 1.0 help search dialog box.

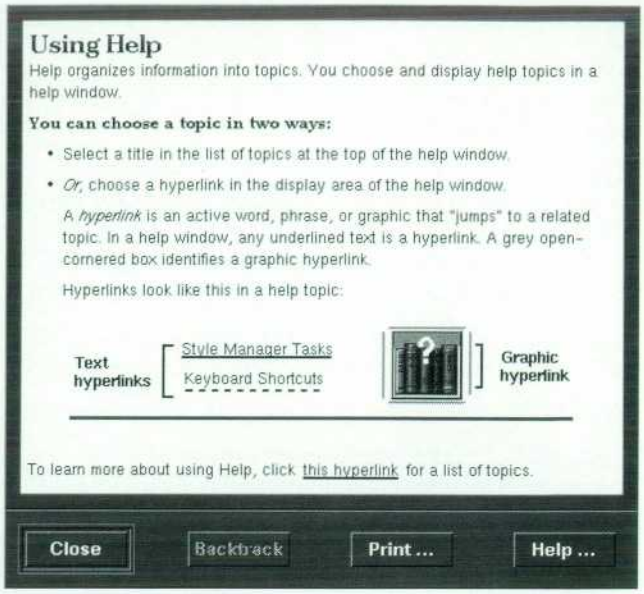


Fig. 5. Quick help dialog box.

functionality is the same as HP VUE 3.0 quick help. It still contains a topic display area and a button box (see Fig. 5).

Display Area

From the developer's perspective the help dialogs are treated as a single widget. They are created, managed, and destroyed as one single object. If one were to look inside one of the help widgets there would be not one monolithic help entity but two separate very distinct components: the text display engine component and the widget component.

The display engine component, as seen by the developer, is the region in the help widget that renders the rich text and graphics and provides hyperlink access and dynamic formatting of the text. The widget component consists of the OSF/Motif code that makes it a widget and the remainder of the user interface, including topic hierarchy, menu bar, and supporting dialogs (print, index search, and history).

Display Area Text. The display area allows text to be rendered using many different fonts. It also allows for various types of text. The author can specify dynamic text that will reformat according to the window size and static text that will not. For dynamic text, a sequence of two or more spaces is compressed into a single space and internal new lines will be changed into a space. For static text, all spaces and internal new lines are honored.

To Reformat or Scroll? While vertical scrolling is accepted as necessary when help topics are longer than the available screen space, horizontal scrolling is not. Therefore, when users resize help windows, dynamic text is reformatted and figures are centered according to the new window size. This dynamic reformatting on the fly is seen as beneficial by the customer.

Scrolling Supported. Even using these line breaking rules, sometimes the lines are too long to fit in the available space. For this case or when static text pushes the boundary of the display area, the help system gives up and displays a scroll

bar so that the user can see the information without having to resize the window.

Flowing Text. The display area has the ability to flow text around a graphic. This is seen as a space-saving measure and highly desired functionality. The graphic can be placed on the left side or the right side of the display area and the text occupies the space to the side of the graphic (see Fig. 6). If the text is too long and does not fit completely in the space beside the graphic, the text wraps to below the graphic.

European (One-Byte) Rules. For most European languages (including English), breaking a line of text at spaces is sufficient. The only other line-breaking rule applied is to hyphens. If a word begins or ends with a hyphen, the hyphen is considered a part of the word. If the hyphen has a nonspace before and after it, it is considered a line breakable character and anything after it is considered safe to place on the next line.

Spaces and hyphens can be tagged by the author as nonbreaking characters. This allows the added flexibility for an author to demand that a word or phrase not be broken.

Asian (Multibyte) Rules. For Asian language support, breaking a line of text on a space is unacceptable since some multibyte languages do not break their words with spaces (e.g., Japanese and Chinese but not Korean). With the Japanese language, the characters are placed one after another without any word-breaking character, since each character is considered a word. There is also the concept that certain characters cannot be the first character on a line or the last character on a line. English (one-byte) characters can be mixed with multibyte characters.

Given these considerations, line breaking for multibyte languages boils down to the following rules:

1. Break on a one-byte space.
2. Break on a hyphen if the character before and after the hyphen is not a space.
3. Break on a one-byte character if it is followed by a multibyte character.
4. Break on a multibyte character if it is followed by a one-byte character.
5. Break between two multibyte characters if the first character can be the last character on a line, and the other character can be the first character on a line.

Rather than hard code the values of those Japanese characters that can't start or end a line into the CDE help system, the message catalogue system is used. This provides a general mechanism for any multibyte language. All the localizers are required to do is determine which characters in their language cannot start or end a line and localize the appropriate NLS file. The file /usr/dt/lib/nls/%l/%t/fmt_tbl.cat contains the values of those characters that are used for rule 5 of the line-breaking rules. If this file does not exist for a given language, rule 5 is ignored and line breaking will occur between any two multibyte characters.

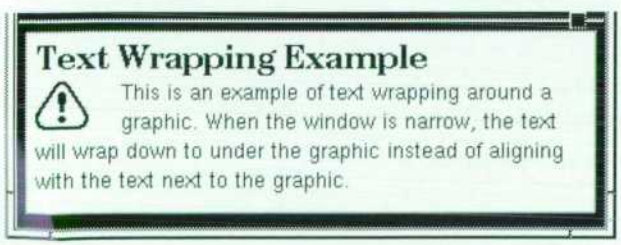


Fig. 6. Example of text flowing around a graphic.

One Library for All Locales. The help system uses the same library for processing one-byte or multibyte documents. It internalizes how many of the rules to use based on the $LANG environment variable and the character set used in the document. If a document specifies an ISO-LATIN1 character set, the display engine does not bother looking at Rules 3, 4, and 5 or using multibyte system routines. Only when the document and the $LANG environment variable specify a multibyte character set are these rules and routines used. This impacts the access and rendering time (minimally) but allows the same library to be used in the United States, Europe, and Asia.

Color Degradation. A feature of HP VUE 3.0 that was strongly desired was the ability to degrade color images. Having to provide three different images for each of three types of files is not feasible because of disk use. The color degradation algorithms used by the CDE 1.0 help system are the same as used by the HP VUE 3.0 help system. For TIFF (.tif) images, the HP image library forces the image to the proper color set depending on the display type. For X pixmaps (.xpm), the Xlib routines take the same approach depending on the display. This leaves the X window (.xwd) files to manipulate.

The task is to reduce an .xwd file from a full-color image to a grayscale image containing no more than the maximum number of gray colors available. The first step in this process maps each of the X Window colors that the image uses to a grayscale luminosity value. This is done using the NTSC (National Television Standards Committee) formula for converting RGB values into a corresponding grayscale value:

$$\text{luminosity} = 0.299 \times \text{red} + 0.587 \times \text{green} + 0.114 \times \text{blue}$$

where red, green, and blue are the X window color values with a range of 0 to 255.

The next step is to count the number of distinct luminosity values used by the image. The help system then determines how many luminosity values to assign to each grayscale. For example, if there are 21 distinct luminosity values and eight gray colors, then the first five gray colors will represent three luminosity values, and the last three gray colors will represent two luminosity values. Next, a gray color is assigned to the luminosity values it will represent. The last step is to change the original color pixel to the gray color its luminosity value indicates.

If the number of distinct luminosity colors is less than the number of gray colors being used, then the help system spreads out the color use. For example, if there are three distinct luminosity values and eight gray colors, then the first, third, and sixth gray colors are used in the image, instead of using the first three gray colors. This provides a rich contrast to the image.

If the system has to degrade the image to black and white, it first calculates the grayscale colors using the luminosity

calculation given above. Then the image is dithered using a Floyd-Steinberg error diffusion algorithm,[3] which incorporates a Stucki error filter.

Color Use. The user can force the help system to use grayscale or black and white by setting the HelpColorUse resource. The CDE 1.0 help system will also degrade an image if it cannot allocate enough color cells for the graphic. For example, if the image uses 31 unique colors, but there are only 30 color cells left, the help system will try to display it in grayscale and if that fails, the image will be dithered to use black and white.

Run-Time Help Volumes

The flexibility and power of this help system are largely placed in the author's hands. With the CDE HelpTag markup language and a creative author, very different and interesting approaches can be taken with respect to presenting information to the end user. Documents can be organized in either a hierarchy with hyperlinks referencing the children at any given level or in the form of a network or web, with a linear collection of topics connected via hyperlinks to related topics. It is up to the author to explore the many capabilities with respect to authoring online help for this system.

Help Volume Structure. A help volume is a collection of related topics that form an online book. Normally, the topics within a volume are arranged in a hierarchy, and when developing application help, there is one help volume per application. However, for complex applications or a collection of related applications, several help volumes might be developed.

Topics within a help volume can be referenced by unique location identifiers that are assigned by the author. Through these location identifiers help information is referenced in the run-time environment.

Help Volume Authoring. The authoring language for the CDE 1.0 help system is HelpTag. This authoring or markup language conforms to a variant of the Standard Generalized Markup Language (SGML (ISO 8879:1986)), which is a simple language consisting of about fifty keywords, or tags.

SGML is a metalanguage used to describe the syntax of markup languages. In a sense, SGML is very similar to the UNIX utility YACC in which the syntax of programming languages such as C is described in a formal, machine readable format. SGML itself does not provide a method for attaching semantics to markup constructs. That is, the equivalents of the YACC actions are not contained in or described by SGML. An SGML syntax description is known as a *document type definition* (DTD).

Other examples of markup languages described via SGML include HTML (HyperText Markup Language), which is used for documents on the World-Wide Web (WWW), DocBook, which is used for documentation of computer software, and PCIS (Pinnacles Component Information Standard), which is used for the exchange of semiconductor data sheets.

SGML is a very powerful markup description language and, as such, allows a great variety in the markup languages to be described. However, in a typical application SGML is used to describe a highly structured markup language with the concept of containment playing a large role.* The concept of containment works very well for applying style to text because text styles can be pushed and popped on entry to and exit from a container.

SGML containers are known as elements. SGML elements are demarcated by the keywords or tags using the form:

```
<keyword> text..... text..... </keyword>
```

where keyword is replaced by the tag name. In SGML terms, the keyword making up the tag name is known as the generic identifier or GI. The form <keyword> is known as the start-tag, and the form </keyword> is known as the end-tag. An SGML element consists of the start-tag and all the text or contained markup up to and including the end-tag. For example, a simple paragraph would be marked up as:

```
<P> This is a paragraph with P being the generic
identifier</P>
```

The syntax of SGML is itself mutable. SGML-conforming documents all start with an explicit or implicit SGML declaration in which SGML features are enabled or disabled for the document, the document character set is specified, and various parts of the SGML syntax are modified for the duration of the document. For example, the default form of an SGML end-tag is left angle bracket, slash, generic identifier, and right angle bracket (<, /, GI, >). For historical reasons, the form of an end-tag in HelpTag is left angle bracket, backslash, generic identifier, right angle bracket (<, \, GI, >). The change of slash to backslash in the end-tag opener is specified in the SGML declaration for HelpTag.

As implied by a previous paragraph, SGML elements may themselves contain other SGML elements. Depending upon the markup described in the document type definition, elements can be recursive.

Further, start-tags can contain attribute and value pairs to parameterize the start-tag. For example, the HelpTag element <list> has an attribute to determine if the list is to be an ordered (number or letter label) list or an unordered (no label) list. Creating an unordered list in HelpTag is done with the following markup:

```
<list>
* item 1
* item 2
* item 3
<\list>
```

which generates:

item 1
item 2
item 3

To create an ordered list (starting with Arabic numerals, by default), one would enter:

```
<list type=order>
* item 1
* item 2
* item 3
<\list>
```

* Containment refers to the hiearchy of an item. For example, a book can contain chapters, chapters can contain sections, and sections can contain paragraphs, lists, and tables. Paragraphs can contain lists and tables, strings can be marked up as proper names, and so on.

which generates:

1. item 1
2. item 2
3. item 3

To create an ordered list starting with uppercase Roman numerals in the labels, one would enter:

```
<list type=order order=uroman>
* item 1
* item 2
* item 3
<\list>
```

which generates:

I. item 1
II. item 2
III. item 3

Note that in the markup described above for list, the individual list items are initiated with the asterisk symbol. In this case, the asterisk has been defined as shorthand or, in SGML terms, a short reference for the full list item markup <item>. Without using short references, the unordered list markup given above as the first example would look like:

```
<list>
<item>item 1<\item>
<item>item 2<\item>
<item>item 3<\item>
<\list>
```

The short reference map of the HelpTag document type definition states that within a list element the asterisk character at the beginning of a line is shorthand for the list item start-tag, <item>, and for second and subsequent list items, also serves as the end-tag for the previous list item.

In HelpTag, the generic identifiers and attribute names are case-insensitive. Attribute values are case-sensitive for strings, but the enumerated values of those attributes with a fixed set of possible values are also case-insensitive.

To reduce typing when creating a help volume, the SGML features known as SHORTTAG and OMITTAG are set to the value yes for the classic HelpTag document type description. The most noticeable result of allowing these features is the ability to leave off the attribute names when specifying attributes in a start-tag. For example, the markup:

```
<list type=order>
```

is equivalent to:

```
<list order>
```

According to the SGML standard, the enumerated values (i.e., order and uroman), must be unique within an element's start-tag so the attribute name can be omitted without creating ambiguity.

Creating a HelpTag Source File. Currently there are no SGML tools for authoring HelpTag documents. To date, all HelpTag authoring has been done using the common UNIX text editors such as emacs and vi. The concept of short references has been used heavily to simplify the process of authoring the HelpTag source and minimizing the amount of typing necessary.

One hindrance to using SGML tools to author HelpTag source code is that the markup language does not adhere strictly to the SGML standard. In particular, short references have been used aggressively to the point of requiring extensions to the SGML syntax. Certain changes in the SGML declaration, while perfectly acceptable according to the standard, are not supported by any SGML tool vendor.

To enhance the possibility of future use of SGML tools for authoring HelpTag source code, a second version of the HelpTag document type definition (DTD) has been created. This second version of the DTD follows the SGML syntax to the letter and makes no use of esoteric features of SGML. This canonical form of the DTD is referred to as the formal HelpTag DTD.

The tools described here for processing HelpTag source code will accept documents conforming to both the classic HelpTag DTD and the formal HelpTag DTD. A switch to the dthelptag script -formal determines whether a document is to be processed according to the classic or formal HelpTag DTD.

The Semantic Delivery Language. The distribution format* chosen for CDE 1.0 changed from the format used in HP VUE 3.0. The old format could not be used for CDE 1.0 because:

- The distribution format was known only within HP and then only by some members of one division. The specification of this distribution format was never published or intended for publication.
- The potential for growth using this distribution format was severely restricted.
- The help volume existed in several files, resulting in problems such as losing one or more of the files during installation.

The run-time format of the CDE 1.0 help system is known as the Semantic Delivery Language, or SDL. SDL conforms to the ISO 8879: 1986 SGML standard. The benefits derived by moving to this distribution format are:

- SDL is based on SGML, which is a standard that is strongly supported and recognized in the desktop publishing arena.
- The format is publicly available. Anyone can create a parser to produce SDL.
- The growth potential of this public distribution format is unbounded.
- The resulting SDL exists in one file, reducing installation problems for developers.

The SDL language can be thought of as a halfway point between the typical SGML application, which ignores formatting in favor of describing the semantic content of a document (from which formatting is to be derived), and the typical page description language such as PostScript™ or nroff, which ignores the semantic content of a document in favor of rigorously describing its appearance.

Unlike typical SGML applications that break a document into specific pieces such as chapters, sections, and subsections, SDL breaks a document into generic pieces known as blocks and forms. SDL blocks contain zero or more paragraphs and SDL forms contain zero or more blocks or other forms (recursively). The SDL block is the basic unit of formatting, and

* The file format of the help files delivered to customers.

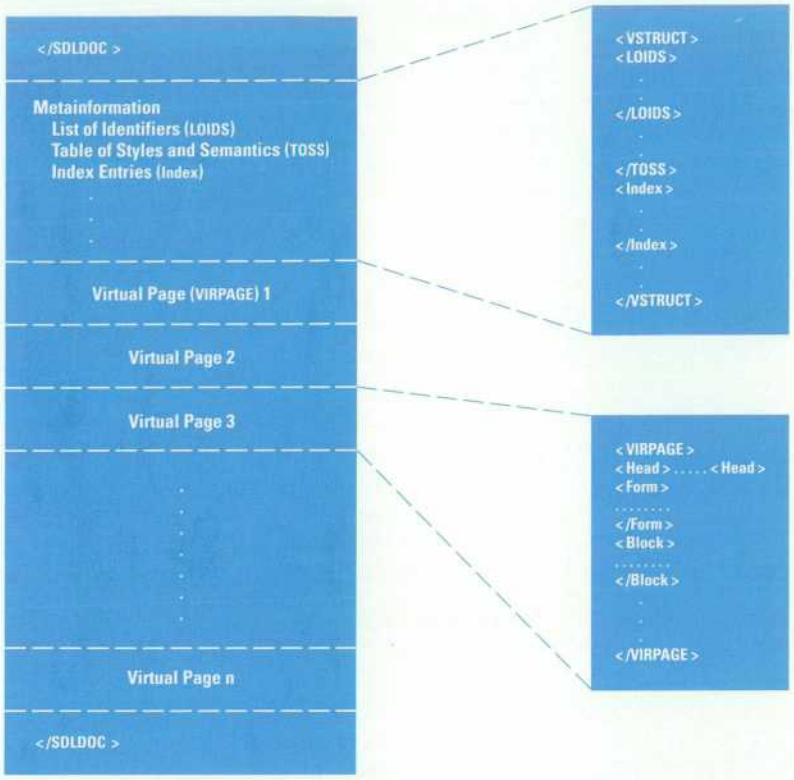


Fig. 7. The structure and elements of an SDL volume.

the SDL form is a two-dimensional array of blocks and forms. Fig. 7 shows the structure and elements that make up an SDL volume, which is also a help volume.

Most elements in SDL have an attribute known as the *source semantic identifier* (SSI), which is used to indicate the meaning in the original source document (HelpTag in this case) of a particular construct that got converted into SDL markup. It is also used to help in the run-time formatting process by enabling searches into an SDL style sheet.

The SDL style sheet mechanism, known as the *table of semantics and styles* (TOSS), is also an element within the SDL document. SDL blocks or forms and their constituent parts can contain a source semantic identifier attribute and other attributes such as CLASS and LEVEL, which are used to match similar attributes on individual style elements in the table of semantics and styles. When the attributes of an element in the body of the document match a style element in the table of semantics and styles, the style specification in that style element is applied to the element in the document proper. That style specification is then inherited by all subelements where appropriate until overridden by another style specification.

Groups of SDL blocks and forms are collected in the SDL construct known as the virtual page (VIRPAGE). Each virtual page corresponds to an individual help topic. For example, the HelpTag elements chapter and s1 through s9 (subchapter levels 1 through 9) would each begin a new virtual page. An SDL virtual page is self-contained in that all the information needed to format that page is contained in the page. There is no need to format the pages preceding a virtual page to set a context for formatting the current page. Fig. 8 shows an example of a VIRPAGE.

For rapid access to help topics, an SDL file also contains an element known as the *list of identifiers* (LOIDS). When an SDL file is accessed and the virtual page containing a specific identifier is requested, the run-time help viewer can scan the list of identifiers to find the requested identifier. Each list entry contains a reference to the identifier it describes and the absolute byte offset from the beginning of the volume to the virtual page containing that identifier.

```
<P>This is a paragraph with a
<KEY CLASS="BOOK"
SSI="TITLE">Book Title
</KEY> in it.</P>
```

(a)

```
<VIRPAGE ID="MYCHAPTER" LEVEL="1"
 DOC-ID="MYDOC">
<HEAD>An Example Chapter</HEAD>
<BLOCK>
<P>This is a paragraph with
<KEY CLASS="BOOK"
SSI="TITLE"> Book Title
</KEY> in it.</P>
</BLOCK>
</VIRPAGE>
```

(b)

Fig. 8. (a) An SDL statement (using SGML syntax) that defines a paragraph with a book title in it. (b) A VIRPAGE representation of the paragraph in (a).

Since SDL virtual pages can be formatted independently of the preceding pages, the information in a list-of-identifiers entry can be used to get directly to the desired page and formatting can begin at that point.

Each of the entries in the list of identifiers also contains an indicator that tells whether the element containing the identifier is a paragraph, block, form, virtual page, or any of the other elements that can carry identifiers. In the case of virtual pages, the list-of-identifier entries also carry an indication of the page's semantic level (e.g., is the virtual page a representation of a chapter, subchapter, etc.).

Duplicating the level and type information of an element in the list of identifiers is a performance enhancement. Knowing which identifiers in the list correspond to virtual pages and knowing the semantic level of those pages allows a human-readable table of contents to be generated for a document by processing only the list of identifiers, avoiding reading and parsing the full document.

Processing a HelpTag Source File. The dthelptag compilation process performs a number of different tasks in generating the compiled run-time help volume:

- Syntax validation
- Conversion from the authored format to run-time format
- Location identifier map generation
- Topic compression.

While designing and implementing the help volume compilation process, techniques for improving performance emerged. The objectives were to create a file that supported quick access and a small overall disk footprint. Fig. 9 shows the components involved in the HelpTag compilation process.

For run-time display, a HelpTag source file must be converted into a form usable by the CDE 1.0 help system viewer. This conversion process is often referred to as compiling or translating the HelpTag source file. The output of the conversion process is a single file containing all the information (except the graphics) necessary to display a help topic. The graphics associated with a topic are left in their own files and are referenced from within the CDE 1.0 help system run-time format, the Semantic Delivery Language.

The dthelptag utility, which converts HelpTag source code to SDL, is a shell script driver for the multiple passes of the conversion process. The conversion takes place in three major steps:

1. The HelpTag source is read in and the first step in conversion to SDL is made. During this step side buffers are created to hold references to graphics and hyperlinks outside of the current document and keyword entries that will later be used to enable keyword searches of the document. The keyword entries correspond to an index in a printed book. Forward cross-reference entries may cause this step to be executed twice for complete resolution.
2. The keyword side buffer is sorted using the collation sequence of the locale of the document. Duplicate keyword entries are merged at this time.
3. The SDL output of the first step is reparsed and examined for possible optimizations. Then the optimizations are performed, the keyword list created in step two and the external reference information from step one are merged in, and the SDL file is preprocessed to facilitate fast run-time parsing and display. Finally, the SDL file is compressed and written to disk.

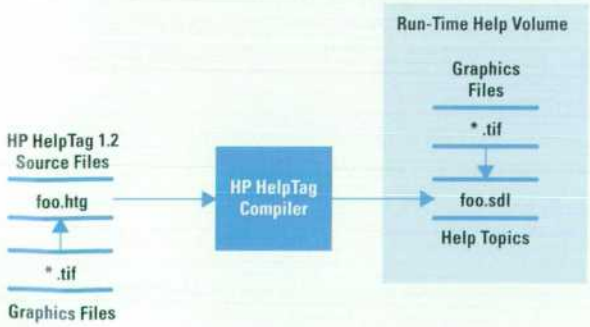


Fig. 9. The HelpTag compilation process.

The optimization possibilities mentioned in step three are created when the first pass must make worst-case assumptions about text to follow or be contained in an element when the start-tag for that element is encountered. If the worst-case scenario does not manifest itself, the resulting SDL code will be suboptimal. The equivalent of a peephole optimizer* is used to detect the suboptimal SDL and to replace, where possible, suboptimal SDL with equivalent but simpler constructs.

The compression mentioned in step three uses the UNIX compress(1) utility, which is a modified version of the LZW (Lempel-Ziv and Welch) compression algorithm. The run-time decompression of the help topic is performed via a library version of the LZW algorithm to avoid requiring the creation of an extra process with its attendant time and memory penalties.

To preserve the random-access nature of the help topics within the help volume, the volume is compressed on a per-virtual-page basis (Fig. 10). That is, each virtual page is compressed and prefaced with a single null byte followed by three bytes containing the size of the virtual page in bytes. After compressing the virtual pages, the list of identifiers must be updated to reflect that all virtual pages following the compressed page are now at a new, lower offset into the file. When the run-time help viewer reads a virtual page, the reader looks at the first byte of the virtual page and if it is a null byte, decompresses the virtual page starting four bytes into the page (the null byte and three bytes for length occupy the first four bytes of the page).

Finally, after all the virtual pages have been compressed, the metainformation in the SDL file, including the list of identifiers, is compressed. Since for performance reasons all the metainformation is at the front of the SDL file, compressing that information must necessarily be an iterative process. After compressing the metainformation, the offsets to the virtual pages in the list of identifiers must be updated to reflect that they all are now at new addresses. When the offsets in the list of identifiers have been updated, the metainformation must be recompressed resulting in new values for the offsets in the list of identifiers.

* In this application, the peephole optimizer reads a small subsection of a document, such as a chapter or paragraph, and works on that portion in isolation from the rest of the document.

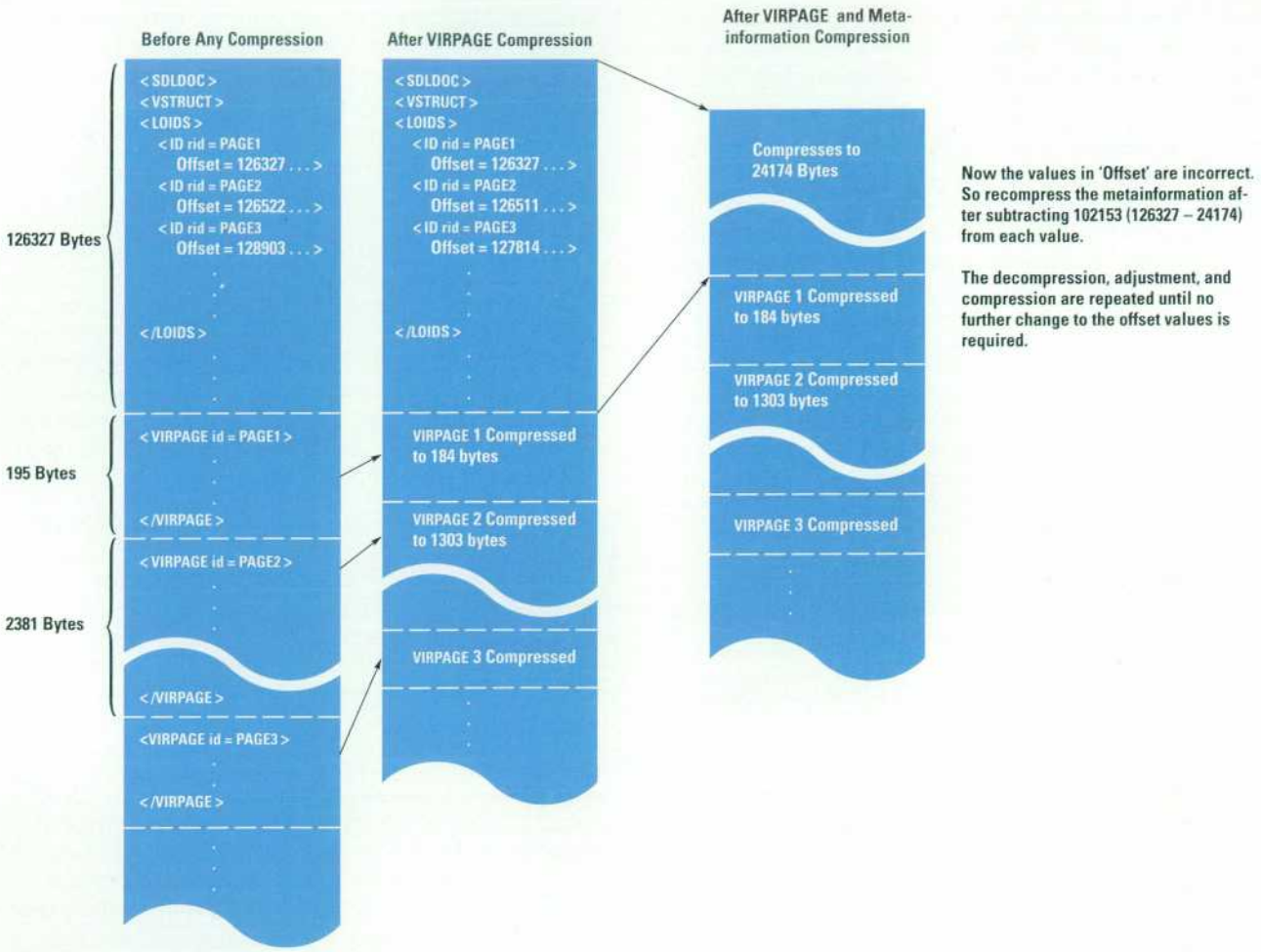


Fig. 10. The compression sequence for an SDL file.

At some point while iterating through this compression and update cycle, the result of compression will be no reduction in size or, worse, an increase in size. If the result is no change, we are done and we can write out the final SDL file. If the result is an increase in size, a padding factor is added after the end of the compressed metainformation and that factor is added to the offsets in the list of identifiers. The iteration then continues, increasing the padding factor on each pass until the size of the compressed metainformation plus all or part of the padding factor stabilizes. The first pass in which the size of the compressed metainformation plus zero or more bytes of the added padding equals the most recently computed offset for the first virtual page terminates the iteration. The compressed metainformation plus as much padding as is necessary is then written to the output SDL file and all the compressed virtual pages follow.

Graphic File Formats. Complaints about the text-only nature of HP VUEhelp 2.0 strongly demonstrated the truth of the adage that "one picture is worth a thousand words." The CDE 1.0 Help System supports the following graphic formats:

- X Bitmaps
- .xwd Files
- .xpm Files
- TIFF 5.0

Graphic Compression. While JPEG compression schemes are common for use with TIFF files, no compression was being used with the X graphical formats. After several complaints from authors about how much space .xwd files require, the help system was modified to find and access compressed files. The author can use the UNIX compress(1) command to compress the graphic files. The help system decompresses the graphic file into a temporary file and then reads the file as usual.

Using compression on X graphic format files can impact access time. For very large graphic images or for a topic that uses many graphics, this impact can be noticeable. The trade-off between speed and disk space is one that the help volume author and application engineer must address. In most cases the best results, both for performance and disk usage, are gained by using JPEG-compressed TIFF images.

Printing

Currently, CDE 1.0 DtHelp renders only text to hard copy devices. The printing solution for the CDE 1.0 help system allows the user to print a comprehensive table of contents and index, complete with page numbers. When an entire help volume is printed, each page is numbered allowing easy cross reference from the table of contents or index. This

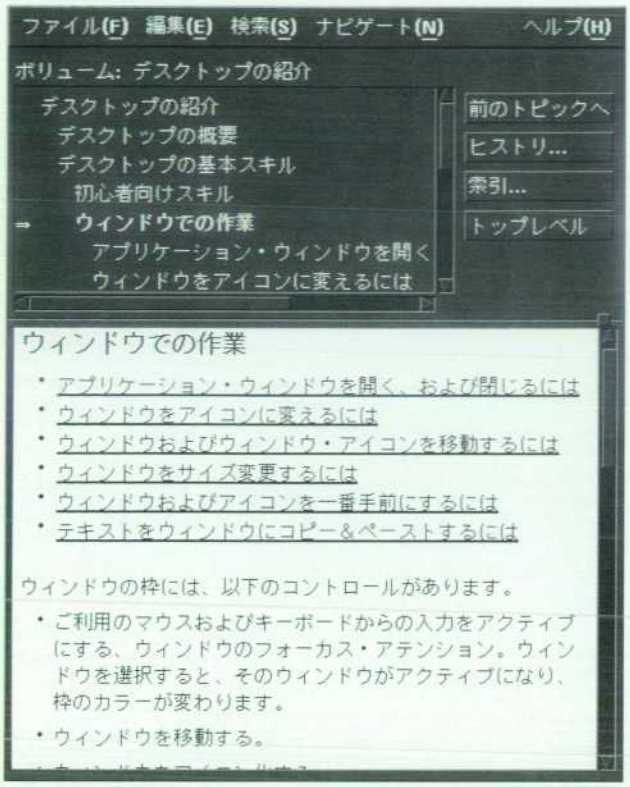


Fig. 11. Sample localized help window.

functionality did not exist in the HP VUE 3.0 help system. It was developed for CDE 1.0.

Localization

The CDE 1.0 help system supports authoring and displaying of online help in virtually any language. The online help information can be authored and translated in either single-byte or multibyte character sets, and all the components within the developer's kit are multibyte-smart and can parse and display the localized information.

The help widgets use the user's $LANG environment variable to determine what language directory to retrieve the requested help volume from. If $LANG=japanese when the request to display help occurs, the widget code will attempt to open the Japanese localized version of that help volume. If one does not exist, then the default version, which is English, will be used.

When an authored help volume is compiled via HelpTag, the author sets the proper character set option. The character set information is used at run time to determine the proper fonts to use for displaying the localized help text. The HelpTag compiler assumes a default locale ($LANG=C). Currently, because of the complexities involved, only one multibyte character set per volume is supported (e.g., a Japanese-to-Korean dictionary cannot be displayed). Fig. 11 shows a sample of a localized window.

Parsing Multibyte Characters. To make dthelptag work for single and multibyte character sets without constantly checking for the length of the current character, all characters are converted to wide characters (wchar_t) on input. This input conversion is driven by a command line option, a HelpTag entity file, or the current setting of the locale. All internal processing of characters is done on wide characters with those wide characters being converted back to multibyte characters on output. Single-byte character sets are treated just like multibyte character sets in that the conversions in and out always take place.

This scheme of doing all internal processing on wide characters has proven to be a very effective means for making one tool work for all languages. The scheme did require implementation of wide character versions of most of the string functions (e.g., strcpy, strlen), but those functions were all quite straightforward to create.

Localizing User Interface Components. The menus, buttons, labels, and error messages that appear in help dialogs also support full localization to native languages. The help dialogs read these strings from a message catalog named DtHelp.cat. Various localized versions are supported by default and included with the developer's kit product. For languages not supplied, the developer must translate the message catalog /usr/dthelp/nls/C/DtHelp.msg and then use the gencat command to create the needed run-time message catalog file.

Conclusion

Building upon the strong foundation provided by the HP VUE 3.0 help system, the CDE 1.0 help system (DtHelp) has become the standard help system for the desktop of choice for the providers of a majority of UNIX systems across the world. As more companies provide the CDE desktop, this help system will become even more pervasive.

References

1. *Hewlett-Packard Journal*, Vol. 45, no. 2, April 1994.
2. *OSF/Motif Style Guide Release 2.1*, Prentice Hall, 1993.
3. R. Ulichney, *Digital Halftoning*, MIT Press, 1988, Chapter 8.

Managing a Multicompany Software Development Project

The development of the Common Desktop Environment version 1.0 involved a joint engineering project between four companies that normally compete in the marketplace.

by Robert M. Miller

In March of 1993 executives from HP, IBM, Sun Microsystems and USL (now Novell) announced at a UNIFORUM conference the COSE (Common Operating Software Environment) initiative. The purpose of the initiative was to unify the UNIX® operating system industry in the areas of distributing computing (ONC, ONC+, DCE),* multimedia technology, object technology, graphics, system management, and desktop technology. While there have been other attempts to unify aspects of the UNIX world, COSE was different in that it succeeded in producing results. CDE (Common Desktop Environment) was the first promised deliverable of this initiative and after two years its completion was announced at UNIFORUM in March of 1995.

CDE 1.0 was a joint engineering effort between HP, IBM, Sun Microsystems, and Novell that was aimed at producing a de facto and de jure standard in a critical and highly visible technology—the X Windows graphical desktop. CDE 1.0 is about 1.2 million lines of source code and over a thousand pages of end-user, system-administrator, and software-developer documentation. The jointly owned CDE 1.0 source tree also contains over 1200 automated regression tests written in C. These tests were developed to test the CDE code base.

The primary aims of all of the participants involved in creating CDE were:
To end the many years of battles between the Motif and OpenLook** camps, which helped to fragment the UNIX industry and slowed the development of UNIX applications
To create a single desktop standard that would provide a common look and feel for users, a common set of administration tools for system administrators, and a common set of desktop APIs for software developers (A list of these APIs is provided in Appendix A on page 11.)
To lay the foundation for the future evolution of the UNIX desktop in a way that would meet the needs of the open systems market.

The original goal was to take the best technologies from each of the four companies mentioned above with roughly 90% of the code coming from existing products and 10% coming from new work. As it turned out this was not possible, and the percentage is approximately 60% reuse and 40% new code.

This was truly a joint engineering project between four large organizations, which required a whole new software development infrastructure and many new processes. This project was also the collision of four different corporate cultures and standards of practice on project management and software development processes. This article examines some of the challenges that needed to be surmounted, and the processes that needed to be created before CDE could be successful.

History

The odyssey of CDE began in the summer of 1992 when HP and IBM began exploratory discussions on the possibility of creating a standard UNIX environment. In the months that followed they were joined by Sun Microsystems and the UNIX Software Labs (USL), which was later acquired by Novell. The agreement between these companies to work together marked the birth of the Common Operating System Environment (COSE) , and more specifically, the Common Desktop Environment (CDE).

However, on the CDE side, agreeing to work together may have been the easiest part of the process. Deciding what technology fragments would be the starting point, what would be the minimum acceptable functionality, and who was responsible for what pieces took a significant amount of difficult discussion. As we worked through these and other difficult issues it became very obvious that good communication would be what would make CDE successful. To facilitate this communication and to get things going, the following teams (with a representative from each company) were established:

Management. The management team dealt with contract issues and major allocation of resources (like funding for localization). They were the oversight team and all other committees reported periodically to the management team. Formal project checkpoints were presented to this team and when necessary, they acted as the arbiter for other teams when a deadlock occurred.

Product Description and Requirements. This team was the day-to-day project management team, handling issues such as staffing, technology ownership, and blending the technologies offered from each company in the new UNIX desktop. They were responsible for finding owners for new or unplanned work items and defining the content and scope of the project.

* ONC is Open Network Computing and DCE is Distributed Computing Environment.

** OpenLook is Sun Microsystem's X Window System toolkit.

Process. The process team was the heart of the CDE project. It was the job of the process team to determine the software life cycle and defect tracking system to use and the mechanism for tracking and reporting schedule information. They defined the hardware configurations that needed to be exchanged between each company. Their most challenging task was to figure out how to allow access to a single, live source code tree for each partner through their corporate firewalls without violating security constraints. This team was essentially responsible for creating a whole new set of policies that would allow four engineering teams from four different companies to work together productively.

Architecture. While the intent was to use as much existing technology as possible, it was important to determine how that technology would be joined together and then identify what new functionality would be needed to ensure a clean, well-integrated desktop. It was the job of the architecture team to ensure that these things happened and to ensure that individual component owners provided the minimum feature set required by each of the four companies.

Standards. From the beginning it was understood that the CDE project would work closely with X/Open® to submit design documentation and later specifications to help CDE quickly become an industry standard. The standards team was responsible for working with X/Open and other standards bodies to understand their needs and processes and to ensure that we could present our materials in a useful way. This team also acted as a channel for feedback to the other CDE teams.

User Model. Each partner in the CDE project had a significant interest in defining the direction of the user model for the desktop (i.e., its look and feel). IBM had CUA (Common User Access) and each of the other participants had been shipping desktop products with fully defined user model policies. The goal of this team was to resolve disagreements about the general user model, as well as deciding specific questions about component behavior and its level of integration with the system.

As this first set of six teams began to interact and started to deal with the issues, it quickly became evident that we needed more teams to focus on specific issues. Thus, the following teams were added:

Build. This team was responsible for keeping the multiplatform builds healthy at each site and making sure that the code was built successfully on a nightly basis.

Learning Products. This team decided what online help and hardcopy manuals would be written, who would write manuals, what tools would be used, the schedule for manual review, and so on.

Performance. Those of us shipping large UNIX desktops had a wealth of experience which showed the need to be very mindful of performance issues early, when it was still possible to correct performance problems. This team was responsible for establishing benchmarks (i.e., user tasks to be measured), finding bottlenecks in the system using performance tools, and working with the individual component owners to fix problems.

Internationalization. This team was responsible for guiding the development teams to ensure that components were properly written so that they could be localized and that policies were followed so that the CDE code worked on all four platforms and followed industry-standard practices in this area.

Localization. This team managed the process of localizing message catalogs, online help, and hardcopy manuals. They were also responsible for deciding on the subset of target languages, finding translators, getting translations into the source tree, and ensuring that the translations were correct.

Test. This team decided on the test tools to be used, defined and implemented the test scaffold, and created the automated and manual tests that would prove the quality level of the desktop.

Change Control. Halfway through the project, this team was set up to ensure that changes were made to the code in a thoughtful and controlled manner and that the appropriate trade-offs were considered. The change control team was instrumental in ensuring that we were able to meet a variety of delivery dates with stable code. In the final months of the project this team played a critical role in determining what defects would be fixed in CDE 1.0 and what defects and design issues would be postponed until a later release of CDE.

Aside from these teams, many of the component owners set up small intracompany teams to better ensure that they were meeting the needs of the other participants. Examples of this were the teams doing the terminal emulator (DtTerm) and the team doing the desktop services (like drag and drop) who held weekly telephone conferences to ensure that there was acceptance from the other partners on their directions and implementation choices.

All of these teams held weekly phone conferences and exchanged email almost daily—especially the build and process teams. Communication was the most difficult part of this project. Despite the fact that we spent communication resources lavishly, communication breakdowns were the root cause of many of the misunderstandings and functionality problems that arose during the course of the project.

Some teams worked better than others. For example, the process team worked extremely well together throughout much of the project, whereas the test team seemed to have more than its share of problems. Partly, this was caused by the sheer enormity of the test team's task. They were faced with the challenge of creating from scratch a whole test scaffold and infrastructure that would meet the needs of all four participating companies. While API testing is a relatively straightforward process, testing GUI clients is not. Also, the test team was in the unfortunate situation of having to evolve processes as we went along which caused a good deal of rework and other problems. A discussion about CDE testing is provided in the article on page 54.

Those teams that worked best did so because the individuals involved had the time to come to some level of trust and respect for each other. They learned to seek a common ground and attempted to present a common position to the other committees. A surprising but significant problem for several teams was the amount of employee turnover resulting from reassignments and resignation. This always caused a team to go through a reset, which was very time-consuming.

New Processes

Some of the individuals in each participating company were very devoted to their own internal processes. Thus, in creating new joint processes for CDE we often faced severe internal resistance from the engineering teams at all sites. This was partly because we were all operating under very tight schedules and didn't have the time to cope with new ways of doing things. Another factor was that, not only did the teams have to learn new processes, but they also had to defend these new processes within their organizations.

By the end of the project we did evolve an effective methodology for implementing new processes across all four companies. The first step involved communicating to everyone concerned (engineers, project managers, etc.) the new process and why it was selected. This was done well in advance of when we wanted to implement the process to allow for questions and issues to be raised at all levels and for the process to be tuned if needed. Later the process team personally visited and got acceptance from all of the project managers so that they would cooperate in implementing (and enforcing) the new process. Finally, it usually involved some degree of follow-up from the process team to make sure that the teams were fully engaged in the new process.

A Single Source Tree

The time available to accomplish the project was incredibly short, and the split of technologies was very interconnected so that it was not possible to work separately and make code deliveries to each other. We were all dependent on new functionality being created by the other teams and needed that functionality to fulfill our own objectives. Also, all of the partners wanted to be on an equal footing with regard to access to the source tree. It was conceivable that people could be working in the same areas of the tree so we had to prevent multiple people from working on the same code at the same time.

We agreed that we would put together a single shared source tree that would be live, in that when an engineer checked out a file at Sun Microsystems, it was immediately locked to all of the other partner sites. The major obstacles to implementing this were our corporate firewalls and the bandwidth between our sites. After significant research (and a lot of help from our respective telecom departments) we put in place a frame relay system between all sites. This gave us T1 (1.544 Mbits/s) bandwidths. While it took a long time to acquire the hardware and work the issue through our various corporate security departments, the efforts paid off handsomely. This was a critical key success factor that really enabled joint development between the various companies.

To overcome the firewall problem we created a separate suspect network at Corvallis which had a single machine on it. This machine was the keeper of the golden CDE bits. This machine was set up to only allow UNIX socket connections from four specific machines (one from each site). A daemon written in Perl* ran on this machine and handled RCS (revision control system) requests coming in from each site. At each partner site a series of Perl scripts were written which wrapped the RCS functionality. When an engineer did a checkout, the request was funneled through another firewall machine to the CDE source machine, which did the appropriate checkout, encrypted the file, and sent it to the requesting machine where it was decrypted and deposited in the appropriate directory. Each night the CDE source machine created a tarball of the entire source tree and passed it along to each partner site where it was built.

Each of the participating companies provided approximately six machines to each partner for build and test purposes. Every night each company built on all four platforms. Each company had to develop significant expertise in developing software on multiple platforms. This simultaneous cross-platform development was not only a necessity, but also very expensive. As engineers made code changes they were required to build and run the new code on all four platforms. Thus, it took a long time to make changes to the code and check it in. The advantage, of course, was that by ensuring that the code always ran on multiple platforms, we didn't find ourselves in a situation where we had to redesign and rewrite functionality at the end of the project to work within different operating system configurations.

Decision Making

It was clear from the beginning of the project that we would not get anywhere if we used a consensus model for decision making. Therefore, we decided to use essentially the model created by the X Consortium. In this model a representative from a single company became the architect for a technology (e.g., Motif toolkit) and was responsible for designing, documenting, and creating a sample implementation of a component or library. Although the architect was required to be open to input for requirements and needs from the other participants, the final say regarding that technology was the responsibility of the architect.

This was the model we agreed to use and in large part we were successful at using it. However, there were times when we ran afoul of this process and the technical decisions of component owners were challenged by one or more of the other participants. These challenges often were escalated all the way up to the management team where, because they were distanced from the technical problem at hand, it took a significant amount of time to work through the issues. It was recognized by all parties that future efforts would make better progress by defining a single point of authority for making final, binding rulings on technical issues.

Scheduling

When the COSE initiative was announced, the schedule for the CDE project was also announced. This schedule was intended to be very aggressive and also reflected the initial assumption that 90% of CDE would be existing code with only 10% new code. The reality was that the melding of the technologies from each of the participants required extensive rewrites in many areas. Also, the functionality was somewhat fixed since none of the participating companies felt they could end up with a desktop that was less functional than the ones they were currently shipping.

Starting with the desired end point, the process team created a schedule that had alpha, beta, and code complete milestones followed by a final test cycle. We needed to define

* Perl (Practical Extraction Report Language) is a UNIX programming language designed to handle system administrator functions.

exactly what each of us meant by these milestones in terms of functionality, defect status, localization status, and branch flow coverage. We created checklists for component owners to fill out and attempted to normalize the data we received from individual project managers.

These first scheduling efforts were flawed for a number of reasons. The first was that coordinating the efforts of four different companies was very time-consuming. Second, we were unprepared for the scope of new processes we would have to put into place and the amount of time it would take to implement them. In some cases we made the classic software management error of trying to legislate the schedule, functionality, and effort with predictable bad results.

We eventually understood the kind of overhead we were dealing with and were able to create a viable schedule that was used to good effect.

Conclusion

There were hundreds of aspects of the CDE project that could have been discussed. Only those issues which in retrospect seemed most important have been discussed. Participating in a multicompany joint development project is a very challenging affair. It requires a willingness to rethink all of the familiar software development processes and tools and to come up with new solutions from both management and engineering teams. It requires an open mind in listening to what each participant is saying since they may use the same terms but with different contexts and meanings. The importance of communication at all levels cannot be stressed enough. Taking the time to build personal relationships at all levels will pay back dividends over the life of the project.

In the best of circumstances, the work being done by the other partners must be constantly monitored to minimize misunderstandings, ensure that commitments are being met, and help shape the inevitable trade-off decisions that occur during a project. Also, it's important to plan for the fact that disagreements and escalations will happen in the most amicable of projects. While it means some loss of control, creating a neutral single point of authority for resolving these escalations will save enormous amounts of time and help to preserve the necessary good will between the participants.

It's natural to underestimate the time it will take to put new processes in place, to develop software on multiple platforms, and to communicate and work out issues with joint development participants. This tendency to underestimate joint project overhead will also appear in individual engineer's schedule estimates. Of course it is absolutely necessary that the engineering teams have the opportunity to resolve ownership and functionality and do a bottom-up schedule that can then be discussed in terms of trade-offs of functionality, performance, and other features before the project schedule is decided upon.

In most cases the work will be done by engineering teams from a variety of geographic locations. Usually these teams do not know each other—except as the competition. Yet they will need to work productively together to make each other successful. It will almost always be worth the time and money to bring the teams together face to face to build a relationship that will prevail through the inevitable tensions of a joint development project.

Finally, it's important to realize that, despite the problems, joint development can be done. Further, because of the diversity of experience and abilities of the different participants, the end result will be richer and appeal to a broader set of customers.

Design and Development of the CDE 1.0 Test Suite

Testing a product whose parts are being developed in four different environments that have different test tools and test procedures requires setting some rigorous test goals and objectives at the beginning of the project.

by Kristann L. Orton and Paul R. Ritter

The Common Desktop Environment (CDE) test team was given the challenge of designing and organizing the development of an automated regression test suite for CDE 1.0 components. This project contained all the usual problems involved in designing and creating a test suite for a large and complex desktop environment, plus a number of challenges that came up because the development was a joint project between four different companies. The article on page 50 provides some background about the creation of the CDE project and the four companies involved.

Several goals for the tests were developed early in the project that influenced the design of the test suite, the choice of software testing tools, and the implementation of the testing process. The rationale for some of these objectives will be developed in more detail in later parts of this article.

In setting the goals for the test suites, we determined that they had to be:

- Easy to develop. There wasn't a lot of time in the CDE schedule.
- Robust. Tests shouldn't start failing because of a minor visual change, a different font selection, and so on.
- Reliable. Tests should find the defects located in the code, but they should not report a defect when there isn't one.
- Consistent operation. Even though four companies would be developing tests, individuals at each company had to be able to run the entire test suite, not just the parts written at their own site. It was not acceptable to make someone learn four different ways to run tests.
- Consistent design and implementation. At the end of the joint development, personnel at each site would get engineering responsibility for the whole suite, including those portions written at other companies. It was important that the tests be written such that an experienced test engineer at one company could easily understand the internal workings of the tests written at other sites.
- Portable. The test suite not only had to run on each of four reference platforms (one from each company), but also had to be easily ported to other nonreference platforms.
- Maintainable. The test suite was not just for the CDE 1.0 sample implementation, but was to be the basis for company products or later versions of CDE.* It had to be relatively painless to update the tests if they had defects, enhance the test suite for new functionality, and so on.

CDE Components

The CDE components were the software under test (SUT) for this project. From a testing point of view, CDE components can be divided into three types: CDE API (application programming interface) components, CDE GUI (graphical user interface) components, and graphical API components. CDE API components have no effect on the desktop, that is, no visual impact. An example of a CDE API component is the ToolTalk® API.** CDE GUI components present desktop graphics that can be manipulated by the user, resulting in graphical changes on the desktop. Examples of CDE GUI components are the file manager and the icon editor. Graphical API components consist of a library of functions like a standard API, except that calls to these functions usually do result in visual changes on the desktop. Examples of this type of CDE component include the DtHelp library and the DtWidget library.

Tools, Utilities, and Environment

This section describes the tools selected for the test suite, the utilities created to augment the tools, and the structure of the test implementation and operating environment.

Synlib

The Synlib API was one of the most important tools used in the development of the test suite. Synlib is a C-language interface that provides a means to simulate programmatically the actions of a user with a GUI. It also contains features that allow the test program to monitor the state of the desktop, such as watching for windows to appear or disappear, checking the title of a window, or checking other desktop features.

Synlib was used to develop all the tests that required either manipulation of objects on the desktop or verification by checking the state of objects on the desktop. This turned out to be the majority of the tests. The CDE test team chose Synlib as the GUI test tool because:

* After the sample implementation of CDE, each participating company was expected to go off and productize their own CDE desktop.

** ToolTalk is the messaging system used by CDE.

- Synlib is portable. The only requirement for using Synlib is that the server be equipped with either the XTEST or the XTestExtension1 extensions,* which are used by Synlib to do such things as simulate keyboard and mouse events. Synlib was functional on all partner systems.
- Synlib is unencumbered. Synlib was developed at HP and was made available in source code form to each of the CDE partners without charge.
- Test development could begin immediately. Engineers could begin writing tests based on the components' specifications. Since Synlib is not a record-and-playback method, a functioning CDE component is not required for initial test development.
- Synlib reduces dependence upon image capture and comparison. Many of the earlier tools for testing GUI components use a method that depends heavily on capturing correct screen images during an initial phase, then comparing that image to the screen in a later test run. Experience shows that this method is quite fragile and likely to produce many false failure reports. With Synlib, other ways of checking for screen state make the need for image capture and comparison much less important.
- Synlib contains a feature that allows position independent manipulation of desktop items. A special data file called a *focus map* is created that defines the keyboard traversal for setting focus to items within a window (Fig. 1).** With the focus map, the test program can set the keyboard focus to a particular button or text field without needing to know its physical location in the window. Since the focus map is a separate data file that is read by the test program, changes in the component that result in changes in the traversal order can be incorporated into the tests by editing the focus map file. Recompiling the test is not necessary. Since the focus map file is platform independent, there only needs to be one focus map file per platform.

  Items that cannot be reached by keyboard traversal need a position. Synlib features the concept of an object file. This is a separate data file that defines locations (x,y pairs) or regions (rectangles) relative to a window origin (Fig. 2). The mouse pointer can be directed to an item by referring to its location defined in the object file. Any change in a component that changes locations requires that the object file be edited, but the test code remains position independent. Also, since the locations vary somewhat for each platform, there needs to be one object file for each platform.

The Synlib test tool is described in the article on page 62.

The Test Environment Toolkit

The test environment toolkit (TET) was chosen as the test harness for the CDE test suite. C-language or shell-based tests were installed in the toolkit and run using the toolkit's utilities. TET is maintained by the X Consortium and is available in the public domain, is portable across many platforms, and has a fairly large group of users. In the X world TET is becoming a de facto standard for test suites.

TET brought two important pieces of functionality to the CDE project. First, it provided an API for journaling, which gave us a standard way to report test results. This was an important aspect in keeping the tests consistent because no matter which site wrote the test, the results journal would have the same format and could be interpreted and summarized using the same tools.

The second piece of functionality provided by TET was a mechanism whereby tests could be collected into groups (called scenarios) and run as a set with a single command. Since the complete test suite for CDE contains literally thousands of individual tests, the ability to run a batch of tests in a single step was essential.

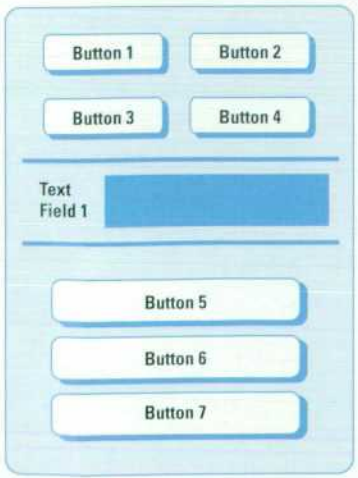
Test Framework

The selection of TET as the framework toolkit for our test environment meant that we had to choose a specific structure for our test tree. This gave us a starting point for building our test suite and provided a logical structure that was expandable and at the same time, robust and maintainable. Fig. 3 shows our test framework.

Located at the root of the tree were the test and environment configuration scripts and libraries containing common APIs. These utilities and libraries were available for use by all of the test suites and, with a few exceptions (Synlib and TET), were developed jointly by the CDE test team. Each function

* XTest and XTestExtension1 are industry-standard extensions to the X Server, which allow a client access to server information.

** Keyboard focus is the term used to describe which of possibly several objects in a window will receive the results of keystrokes. For example, a Motif window may have three different text fields, and the field that has the keyboard focus will get the characters typed by the user.

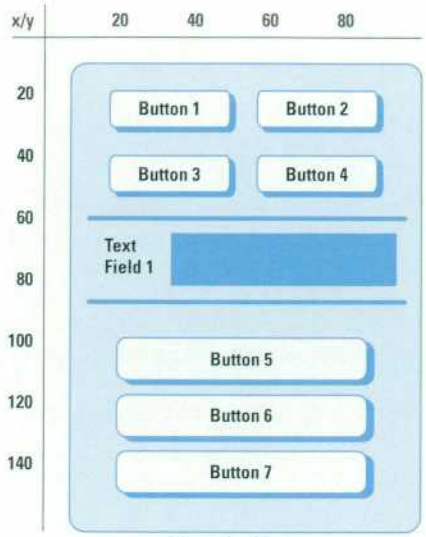


Motif GUI Window

```
(FocusMap MotifGui
  (FocusGroup One
    (Button1 Button2 Button3 Button4))
  (FocusGroup Two
    (TextField1))
  (FocusGroup Three
    (Button5 Button6 Button7)))
```

Focus Map

Fig. 1. An example of a focus map. For keyboard traversal purposes, the Motif client can organize its objects in focus groups. In this example there are three focus groups. The focus map file allows the test designer to name the objects that can receive input focus and define the correct combination of keystrokes (tabs and up and down arrows) needed to shift the focus from one object to another. For example, the test designer can specify that the input focus should be moved to object MotifGui.Three.Button6 without any information about the location of the object.



Motif GUI Window

```
Location Button1 20 20
Location Button2 60 20
Location Button3 20 40
Location Button4 60 40
Region TextField1 37 62 90 80
Location Button5 20 100
Location Button6 20 120
Location Button7 20 140
```

Object File

Fig. 2. An example of an object file. The test designer can use the location in the object file to set the focus to Button6 as is done in Fig. 1. However, if the GUI is redesigned and objects are moved, the test will fail until the object file locations are updated.

and binary had to have, at a minimum, an associated man page describing its functionality and use.

The following sections describe the test tree components in Fig. 3 as they were used for testing CDE 1.0.

Libraries. Synlib and DtTest are described in more detail in other sections of this article. The tet_api library was modified slightly for our environment, adding code that allowed the tests to be run from a build tree. The area occupied by the build tree was not writable by the test user, so any tests could not expect to use that space for creating test run data.

Utilities. The utilities were important for providing a common way for tests to set up and clean up their environment. The build_scen script was used to create a new TET scenario file by searching through the test files for tet_testlist structures, preventing the tests and their associated scenarios from getting out of synchronization. The TET journal filter searched through the usually huge journal file and separated failures and their associated assertions, improving the efficiency of test result analysis.

Dt Config. The scripts in the Dt Config directories contained environment variables used by TET and Synlib and for satisfying platform-specific environment requirements. There was a convenience function in the DtTest library that set up global pointers to these environment variables, making them accessible from any test.

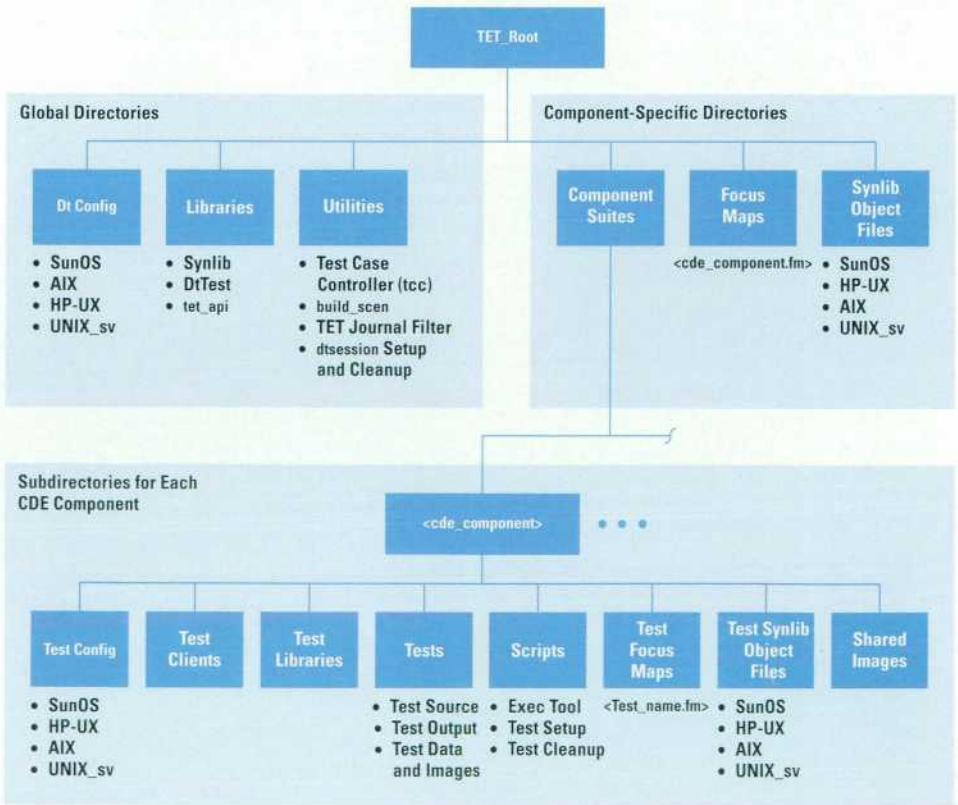


Fig. 3. Shared test tree framework.

The other pieces of the tree are specific to each component test suite. A basic map of each component lives at the top of the tree, providing maximum leverage for suites testing inter-client communication and doing any system testing.

Focus Maps. Each CDE component having a GUI front end was required to provide a focus map file that was accessible to all the test suites. For example, the calculator component had a file in the directory named dtcalc.fm containing the keyboard focus map for the calculator GUI.

Synlib Object Files. Each CDE component provided an object file containing aliases for the interesting x,y locations in its GUI. Since these objects were platform dependent and varied greatly from platform to platform, object files had to be generated for each of the reference platforms.

Component Suites. There was a subdirectory under comp_suites for each GUI and library in CDE. For most suites, this was the top of the test tree branch. For suites having functionality that was more complex than could be encompassed in one suite, this directory contained several subdirectories for better granularity of the covered functionality. A good example of this is the CDE help component, which is made up of a help viewer and several APIs. This suite was broken into four subsuites with a top-level library and test client that was accessible by all suites (Fig. 4).

Each test suite directory had a regimented structure so that the suites could be efficiently maintained in a consistent manner. At the same time, we tried to provide some flexibility so that creative solutions could be engineered for the myriad of difficulties faced by the test-suite developers.

Test Libraries. As the test developers learned the test tools and framework, they found a lot of commonalties among the tests in their suites. These commonalities were gathered into a common library so they could be used by other tests. Some of these commonalities included the way tests were started and closed, and the verification methods used on components.

Test Clients. Many of the suites that tested the CDE APIs used a test client to exercise their code and verify their assertions. For example, to test the subprocess control daemon, a couple of clients were developed to send and receive messages, verifying the validity of data sent at the same time.

Test Config. Like the Dt Config scripts described above, Test Config scripts set up environment variables that were optionally platform dependent and used by the tests. An associated function was added to the test library to set up environment variables as global variables accessible within their tests in the same manner as that in the DtTest library.

Test Focus Maps. The majority of the test focus maps under this subdirectory were used for defining the keyboard traversals of test clients.

Test Synlib Object Files. Object files contain aliases for x,y locations in GUI components. Test developers used these location aliases to define areas in the GUI that needed to be tested for cut and paste and drag and drop operations and regions for image comparisons.

Shared Images. Although we encouraged test developers to limit the number of image comparisons, this method had to be used when validation was not possible by any other means. The next best thing was to reuse images so that more than one test could use the same image for verification.

Tests. The tests directory contained the tests and other files that made up a test suite. A README file in the directory described the test suite structure and any special environment considerations that needed to be taken into account when running the tests. Optionally, there was a directory at the top level that contained manual test descriptions. The tests were intended for tests in which automation was not feasible (e.g., testing the animation look when a file was dropped on a drop site).

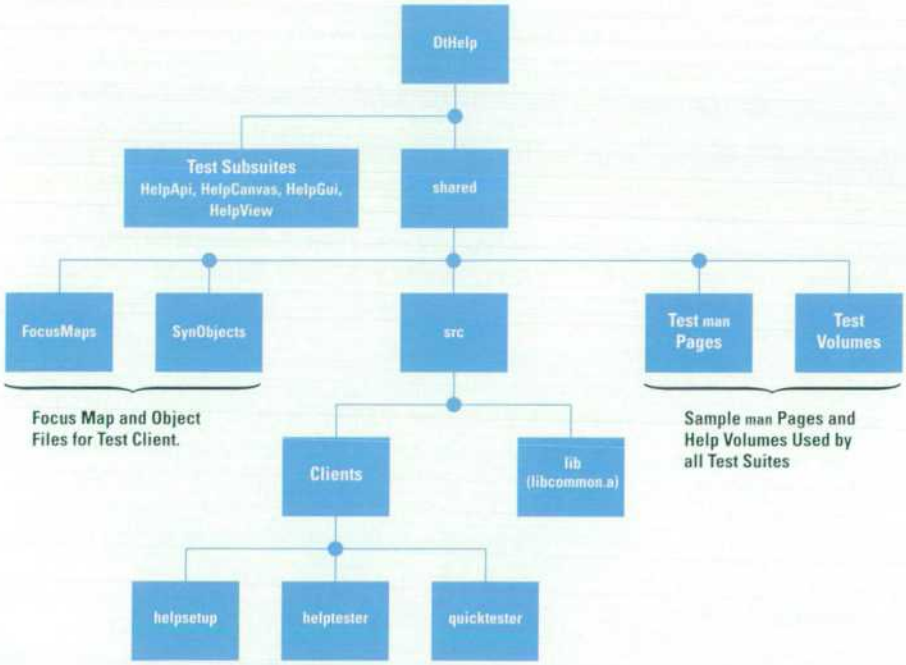


Fig. 4. Component suite for CDE help.

The test source code lived in the tests directory. This directory contained as many subdirectories as needed. Developers were encouraged to divide their tests into groups based on a component's functionality, making it easy to determine which tests covered which piece of code. This directory also contained an image directory for images that applied to only one assertion and a data directory that contained validation data such as window titles and text for text comparisons. The test output directory was a place holder used by the test case controller (tcc) at run time. The tests stored information about failures in these directories and the test case controller copied them into a results directory at run time, facilitating easy results analysis. The following listing is an example of a typical test source file.

```
 #include <DtTest.h> /* DtTest library header */
 /* Forward declaration of routines */
static void tp1();        /* test purpose one*/
static void startup(),cleanup(); /*TET startup
and cleanup routines* /

/* Initialize test environment toolkit data
   structures */
void(*tet_startup)()   = startup();
void (*tet_cleanup)()  = cleanup();
struct tet_testlist[]  = {
    { tp1, 1 },
    { NULL, 0 }
};

static void startup()
{
   DtTestStartup();
   DtTestGetTestEnv();
}
static void cleanup()
{
   DtTestCleanup();
}

static void tp1()
{
   DtTestAssert(1, DT_ACCEPT | DT_REGRESS,
              "Assertion text here.") ;
/* Code for validation of the assertion would go
   here. */
    DtTestResult(TET_PASS,
              "Assertion passed.");
}
```

Scripts. Many of the test suites used the test case controller mechanism that allows the tests to define their own execution tool. This script, found in the scripts subdirectory, did such things as start the component with special options and set test-specific environment variables. Additionally, a test developer could define a separate script for test-specific setup and cleanup that was called from within the exectool.

The DtTest Library

Early in the CDE test development process, it was noted that certain combinations of TET or Synlib functions were commonly used as a unit. This led to the creation of the DtTest library which provided a more efficient mechanism for accessing these units in a uniform manner.

This library is a set of convenience functions developed to supplement the basic TET and Synlib functionality. These functions provided an easier way for test developers to perform common testing practices and support the CDE testing methodology. Use of the DtTest library functions also helped enforce internal consistency among tests developed at different sites.

The convenience functions that augmented TET functionality made it easier to print messages in the journal file, enter the test result, and control the operation and execution of tests. The DtTest library provided an initialization and a cleanup routine that TET did not provide, and a way to get the values of the CDE environment variables.

Functions to supplement Synlib were focused on two areas. One was to deal with text in a GUI text field or text widget. Functions were written to place text, retrieve text, or retrieve and compare text. The second area dealt with techniques for screen image capture, storage, and comparison. These functions took care of image naming conventions, storing images, and saving image analyses when image comparisons failed.

The CMVC Defect Report Mechanism.

The tests were programs, and like any set of programs some of them contained bugs. Often the defects were found not by the originators of the tests, but by other partners during test runs. After some unsuccessful attempts to keep track of defects and defect fixes via email and phone conversations, the test team started using the Code Management and Version Control, or CMVC tool from IBM. This method of reporting, tracking, and verifying defects and fixes was used for all CDE components.

Code Coverage Tools

Code coverage tools provide coverage statistics for a set of tests. They determined how much of the tested component's code was actually exercised during a test run. Typically these tools produce coverage statistics in terms of the number of branches taken, the number of functions called, and so on. The CDE test team needed to get coverage statistics since these metrics were one of the measures of a test suite's completeness.

There was no single code coverage tool that was generally available to all of the CDE partners, but each had a solution that worked on their platform. Analysis of each of the different tools showed that they produced results that were comparable. The test team agreed that all partners would use their own code coverage tools to provide test coverage statistics for their own component test suites. These metrics would be provided at certain developmental milestones to ensure that adequate progress was being made toward the coverage goals.

Test Design Objectives

Early in the project, the test team adopted a set of high-level design objectives. These objectives were based on the testing experience that was brought to the project from the different partners. Although team members from each company had considerable software testing experience, the philosophy and testing methods used by each company were often quite different.

The test team did achieve agreement on a number of basic objectives and goals, which resulted in the following high-level test design goals.

Formal Test Plan and Test Cases. Each CDE component was required to have a formal test plan document, written by the test engineers who had responsibility for that component. The test team provided a template document (see Fig. 5). The test plan was written before test development began. One of the sections of the test plan contained a list of test cases for the component. These test cases were based on the contents of the component's formal specification document. The rationale for this requirement was to get test developers to plan their components' test suite carefully before writing any tests.

Assertion-Based Testing. Tests had to be written using an assertion verification methodology. With this method a test begins with a statement of functionality about the component that can be answered true or false. For example, "Clicking mouse button one on the cancel button will dismiss the dialog box," or "Calling the OpenFile function to open file foo in read mode will return error code NoSuchFile when foo does not exist." Following the statement of assertion in the test comes the code necessary to perform the action implied in the assertion (e.g., click the cancel button, call the OpenFile function). Code to verify the results of the action would report a pass or fail to the journal file.

Testing Based on Component Specifications. The test team strongly encouraged that the design of the initial tests be based on a component's specifications. A component was to be treated as a black box until all the functionality described in the formal specification document was tested. This approach ensured that when a component was out of phase with the specifications, an error would be reported. Resolution of the error sometimes meant updating the specification, ensuring that changes would be captured in the documentation.

Testing Based on Code Coverage Goals. The test team realized that formal specification documents cannot cover every single detail, and that a test suite based only on the document contents would fall short of achieving the test coverage goals. Thus, the team decided that after the functionality covered in the specification document was incorporated into tests, the test suite's code coverage would be measured. If coverage fell short of the coverage goals, the code could be examined to determine how to design new tests that would exercise code branches or functions missed by the initial test set.

Test Suite Completeness Based on Coverage Statistics. A component test suite would be considered complete when it tested all the functionality described in the specifications and reached the minimum coverage goals set by the test team. The initial coverage goals for all components were that 85% of all branches would be hit at least once, and 100% of all internal functions would be called at least once. There was an additional requirement for API components that 100% of all external (developer visible) functions would be called at least once. Typically these statistics would be presented as a triplet of numbers (e.g., 85/100/100).

This was one of the goals that was revisited during the project. Although there was not total agreement in the test team, the majority felt that writing tests for GUI components was more difficult than writing tests for nongraphical API components. As a result, the GUI coverage goal was lowered to 70% overall, with only 10% required in automated tests.

Automated Tests. The objective was to create an automated regression test suite. As the test plan overview document stated, "The expectation is that all tests are automated and that only where absolutely necessary will manual tests be acceptable." However, as noble as this goal was, we had to lower our expectations a bit. By the end of the project the test team was still requiring automated tests for the API and graphical API components, but for GUI components, the requirement was that the automated tests provide at least 40% branch flow coverage. The remaining 60% branch coverage could be obtained by manually executing tests based on either the list of test cases in the test plan or in a separate test checklist document.

Minimum Reliance on Screen Image Capture and Comparison. Experience with automated testing of desktop components showed that test suites that relied heavily on screen image capture and comparison were found to be unreliable—they would generate many false failures. The new Synlib test tool contained features that could be used to verify a correct desktop state without resorting to comparisons to previously saved "golden" images, and the test team aggressively investigated and promoted these techniques. It's estimated that the CDE test suite contains less than 10% of the "golden" images that would be required if a record-and-playback tool had been used.

Common Desktop Environment
Functional Verification Test Plan

1.0 Introduction — Define the scope of the testing and the components tested.

2.0 Test Environment — Define hardware and software requirements.

3.0 Test Matrix
- 3.1 Test Strategy — Plan of attack, including the functionality covered.
- 3.2 Test Cases — List of test cases, identifying interoperability, I18N, stress, and interplatform tests.
- 3.3 Untested Code

Fig. 5. Outline for a test plan.

Test Development Process

Once the CDE test team had straightened out the details of the testing framework and produced a set of guidelines for the test developers to follow, it was time to implement the solution. We set up test goals for each of the component development milestones. We jointly developed a training course designed to get test developers familiar with our process and test tools as soon as possible. Lastly, we set up a process for each of the members on our team to follow as we accepted the test suites for incorporation into the CDE sample implementation test suite.

Each CDE component had functionally complete, functionally stable, and code complete milestones. Each of these

milestones had test coverage goals as shown in Table I. An API would be functionally stable when it had 70% coverage.

Table I
Test Coverage at Component Milestones

Milestone	API	GUI	External (Breadth)	Internal (Depth)
Functionally Complete	50%	5t*/10%	90%	70%
Functionally Stable	70%	40%	100%	90%
Code Complete	85%	70%	100%	100%

* Five tests defined or 10% coverage, whichever is greater.

When measuring the source code coverage of a test suite, besides the base number of overall coverage (lines of code hit divided by the total lines of code—the numbers in the GUI and API columns in Table I), there is also a need to look at the external or breadth coverage. For an API, this was the number of external functions called divided by the total number of external functions. For a GUI, this was the number of functional modules exercised divided by the total number of modules (i.e., Were all the buttons in the dialog box hit?). Internal or depth coverage is the number of internal (nonpublic) functions called divided by the total number of internal functions.

All components were expected to have a base level of automated acceptance tests, but GUIs could make up the rest of their test coverage through either automated tests or well-defined manual tests. For some components, such as the front panel which had to test the animation of the subpanels, creating some scripts that would be run by hand and manual verification of the results was the best way to go. APIs were expected to provide all automated tests.

As components reached one of their milestones, the test team representative from the company responsible for the component would check for more test development specific milestones. These milestones were put in place to ensure that the test team's final product was a robust, maintainable test suite.

The first item the test team checked for at each milestone was a complete test plan. It was important that the test plan thoroughly define the testing to be done since as a last resort, we would use the list of test cases in the test plan as a list of manual tests to complement the automated tests.

The second task the test team performed at the milestones was a review of the automated tests. We developed a criteria checklist that would check for the following:

- Acceptance tests. The priority for the functionally complete milestone was to have automated acceptance tests that could run in less than an hour.
- Assertions. Assertions were checked to be sure that they described the component's functionality and clearly stated what was being tested and how it was being verified. This was the hardest area for new test developers to learn and the hardest for us to judge.
- Use of the DtTest convenience functions. These functions were developed to ensure consistent journaling of tests, error handling, and naming conventions of image and data files.
- Use of copyright notices and standard header files. These tests were done manually for the first five or ten tests for a particular component. The test developers were expected to use their "blessed" test suites as a template for the rest of their tests.
- Test suites must run autonomously. TET allows for a very fine granularity of execution, down to individual test purposes within one test case. A test purpose is a function made up of an assertion, code validating the assertion, and code for reporting the result. Test purposes could be grouped together within an invocable component, ensuring that they always ran together, but beyond that, these invocable components always had to be able to run on their own.
- Test-specific libraries and clients. These were reviewed to be sure that the library calls were documented in a header file and test clients in a README file.
- Portability. Tests were reviewed for nonportable practices such as hardcoded path names and x,y locations. The total number of stored images was also expected to stay under 15 per test suite.
- Test Execution. Tests had to run using the test case controller on every platform.

As the project wore on, this checklist stretched out to 77 items and became affectionately known by the component engineers as the "77 points of pain." Our last milestone item was to check that memory integrity was being checked.

About halfway through the project, test development efforts really got into full swing at all the companies. We all used temporary test engineers from time to time, and it was necessary for the test team to get these new engineers familiar with our methodologies as soon as possible. We jointly developed a two-to-three-day training course that new test engineers took before getting started. This covered training for the tools, how to write a good assertion, and creating Imakefiles. By the third day, a test engineer would have completed at least one test and be familiar enough with the test tree structure to get around without help. We used some test suites that were good examples of the kind of tests we were looking for, and we had an example test suite as a guide for engineers to use for doing more complex functional verification. Finally, we developed a "how to" document that architecturally described the test tree design and defined all of the tools and interfaces available for use. Despite our best efforts, it still took about two to four weeks for a new test engineer to develop the ability to do a couple of test cases per day.

Test Execution Cycles

Throughout the development of CDE there were several places where we stopped and executed a common test execution cycle in which all CDE partners participated. These test cycles were driven by events such as conferences and trade shows, where the desktop was displayed, and milestone completion dates. We developed a test report form so that the results could be compiled in a consistent fashion and results reported at each company. Journal files from the test runs were checked into the tree so that it was easy to check for both component and test regressions. After each

test cycle, we would do a postmortem to improve the process for the next test cycle.

Our first goal in the test execution arena was to define an execution cycle that would enable each company to execute the tests in a uniform, automated fashion. The phases of this cycle are listed below. Phases four through eight were repeated for each test suite.

Phase I Machine Setup. This included setting up the reference platforms at each company, taking into account any hardware or software requirements documented in the test plans.

Phase II Test Environment Build and Installation. This was easy for us since our test trees were set up to build on a nightly basis in the same fashion as the source trees. The more difficult part was the installation. Since the test suites would certainly run overnight, the build would interrupt the test execution, causing indeterminate results. The short-term solution was to turn off test builds during test cycles. For the long term, we wanted to create an installation process as automated as the build process.

Phase III General Test Environment Configuration. This phase included defining configuration data and executing any setup programs, including:

- Putting the general environment configuration defined in the Dt Config files into the test execution environment
- Setting up user permissions and scripts that must be run as root.

Phase IV Component Specific Test Environment Configuration. Analogous to the previous phase, during this phase the component's test environment was set up, including:

- Putting the component-specific configuration file into the test execution environment
- Starting the session using the dtsession setup script
- Running build_scen to create a current scenario file.

Phase V Run Test Cases. The tests were executed using the test case controller, specifying an output directory for saving results and a journal file.

Phase VI Test Results Evaluation. The journal files were run through the TET journal filter script to find any test failures.

Phase VII Component-Specific Shutdown. Between each test suite, the test environment was cleaned up using the same script as for setup. The session was stopped via the cleanup dtsession script to keep the previous tests run's exit state from affecting the next test.

Each company had to check their test results into the shared test source tree at the end of the test cycle. They had to state for each component the type of testing done (automated or manual), the current branch flow numbers, the number of test cases run (number of test assertions), and the pass/fail status for the total run. A postmortem was done after the test cycle, looking in particular for test suites that had different outcomes at the different companies. Defects were filed against the test suites and the components, and if a particular test cycle was targeting a release, the defects were fixed and the tests were rerun.

System testing was done more sporadically. We developed a fairly extensive system test plan covering areas of interoperability, I18N (internationalization), interplatform, and stress testing. Unfortunately, these tests were never automated, in part because of a shortage of resources. These tests were more complex, and therefore more difficult to automate than the functional tests. We did make sure that both interoperability and I18N functionality were tested with each test cycle. We usually relied on the CDE test team members to manually run through the system test plan for their company's platform. For interoperability, a matrix was developed showing the types of interclient communications that were allowed for each component. The I18N section described pieces of the component that used input methods for EUC (Extended UNIX® Code) 4-byte characters as well as sections that were expected to be localized. Our reference languages were Japanese and German, so the manual checklist was run against these two languages.

Conclusion

By the time the CDE sample was done, some of the CDE test suite was not complete. Some of the components had no automated tests and some test suites were in various states of completion. However, the existing test suites and the test framework provided a good basis for a maintainable test suite for the HP CDE product. In fact, the framework and methodologies have been expanded to encompass other HP products with a great deal of success. Work continues to be done in this area, adding further expansions such as internationalized tests and other system testing.

Synlib: The Core of CDE Tests

Synlib is an application program interface for creating tests for graphical user interface applications. A collection of Synlib programs, each designed to verify a specific property of the target software, forms a test suite for the application. Synlib tests can be completely platform independent—an advantage for testing the Common Desktop Environment (CDE), which runs on the platforms of the four participating companies.

by Sankar L. Chakrabarti

Synlib is a C-language application program interface (API) designed to enable the user to create tests for graphical user interface (GUI) software. The user simply tells Synlib what to do and it will execute the user-specified tests on a variety of displays and in a variety of execution environments.

A complete description of the Synlib API is beyond the scope of this article and can be found in references 1 through 4. In this article we will only indicate how the ideas and capabilities supported by Synlib were applied to the development of tests for the Common Desktop Environment (CDE) described in the article on page 6. An overview of CDE test technology including a description of the role played by Synlib can be found in the article on page 54.

To test GUI software, we create programs using the Synlib API. These programs in turn manipulate and verify the appearance and behavior of the GUI application of interest (the target application). The recommended approach is to create many small Synlib programs to test the target software:

1. View the target GUI software as a collection of implemented properties.
2. Create a Synlib-based program to verify each property.
3. A collection of such Synlib programs forms a test suite for the GUI software.

With Synlib, the main task of testing GUI software reduces to creating the individual test programs. Considering each test program to be your agent, your task is to tell the agent what to do to verify a specific property of the program you wish to test. Assume that you want to test the following property of the front panel on the CDE display (Fig. 1): *On clicking the left mouse button on the style manager icon on the CDE front panel, the style manager application will be launched.* It is likely that you will tell your agent the following to verify if the property is valid for a given implementation of the front panel:

1. Make sure that the front panel window is displayed on the display.
2. Click the style manager icon on the front panel window. Alternatively, you might ask the agent to select the style manager icon on the front panel and leave it to the agent to decide how to select the icon. Synlib supports both alternatives.
3. Make sure that a new style manager window is now displayed. If the style manager window is mapped then the agent should report that the test passed. Otherwise, the agent should report that the test failed.

These instructions are captured in the following Synlib program. All function calls with the prefix Syn are part of the Synlib API.

```
main(argc, argv)
int argc;
char **argv;
{
 Display *dpy;
 int windowCount;
 Window *windowList;
 Window shell_window;
 char *title;
 SynStatus result;

 dpy = SynOpenDisplay(NULL);
 result = SynNameWindowByTitle (dpy,
  "MyFrontPanel", "One", PARTIAL_MATCH,
  &windowCount, &windowList);
if (result == SYN_SUCCESS)
 {
  result = SynClickButton(dpy, "Button1",
   "MyFrontPanel", "styleManager_icon");
  /*
   * In the alternate implementation using focus
   *  maps we could simply say:
   * result = SynSelectItem (dpy,
   *  "MyFrontPanel",
   *  "StyleManager_Icon_FocusPath");
   */
  result = SynWaitWindowMap (dpy,
    "StyleManager",TIME_OUT);
```

Fig. 1. Front panel of the CDE desktop.

```
if (result == SYN_SUCCESS)
 { result = SynGetShellAndTitle (dpy,
   "Style Manager", &shell_window, &title);
    if (strcmp (title, "Style Manager") ==
    0)
     printf ("Test Passed: Style Manager
     window appeared. \n");
  else
     printf ("Test Failed: Expected Style
      Manager window; found %s\n",title);
 }
 else
  printf ("Test Failed: Expected Style Manager
    window would map but none did.\n");
}
else
 printf ("Test Aborted: Front Panel window
  could not be found. \n");
}
```

SynOpenDisplay() connects to the display on which the target application is running or will run. SynNameWindowByTitle() determines if a window of the specified title (in this case One) is already displayed. If so, the user (i.e., the programmer) chooses to name it MyFrontPanel. Through SynClickButton(), the Synlib agent is instructed to click on the window MyFrontPanel at a location called styleManager_icon. The agent is being asked to expect and wait for a window to map on the display. The function SynWaitWindowMap accomplishes the wait and names the window StyleManager if and when a window maps on the display. If the mapped window has the expected title, StyleManager, then the agent is to conclude that the test succeeded, that is, the front panel window had the specified property (clicking on the style manager icon really launched the style manager application).

In practice, the tests would be more complicated than this example. Nonetheless, this simple program illustrates the basic paradigm for creating test agents. You need to name the GUI objects of interest and provide a mechanism for the agent to identify these objects during execution. The agent should be able to manipulate the named objects by delivering keystrokes or button inputs. Finally, the agent should be able to verify if specified things happened as a result of processing the delivered inputs. Synlib is designed to provide all of these capabilities. Table I is a summary of Synlib's capabilities.

Table 1
Synlib Capabilities

Functions to Name GUI Objects of Interest
- SynNameWindow
- SynNameWindowByTitle
- SynNameLocation
- SynNameRegion
- .
- .
- .

Functions to Deliver Inputs to Named Objects
- SynClickButton
- SynClickKey
- SynPressAndHoldButton
- SynReleaseButton
- SynMovePointer
- SynPrintString
- SynPressAndHoldKey
- SynReleaseKey
- SynSetFocus
- SynSelectItem

Functions to Synchronize Application State with Test Agent
- SynWaitWindowMap
- SynWaitWindowUnmap
- SynWaitWindowConfigure
- SynWaitProperty
- .
- .
- .

Functions to Verify the State of a GUI Application
- SynGetShellAndTitle
- SynStoreText
- SynCompareWindowImage
- .
- .
- .

Miscellaneous Functions to Process Needed Test Resources from the Environment
- SynParseCommandOptions
- SynParseObjectFile
- SynBuildFocusMap
- SynParseKeyMap

Platform Independence

Synlib programs can be written so that they are completely platform independent. For example, in the program above there is nothing that is tied to specific features of a platform or a display on which the target GUI application may be executing. All platform dependent information can be (and is encouraged to be) abstracted away through the mechanism of *soft coding*. In the program above, the statement using the function SynClickButton is an example of soft coding. The last parameter in this statement, styleManager_icon, refers to a location in the front panel window. The exact definition of the location—the window's x,y location with respect to the FrontPanel window—is not present in the program but is declared in a file called the *object map*. At execution time the object map is made available to the Synlib agent through a command line option. The agent consults the object map to decode the exact location of the named object styleManager_icon, then drives the mouse to the decoded location and presses the button. Because the location is soft coded, the program itself remains unchanged even if the front panel configuration changes or if the exact location of the named object is different on a different display. The named location, styleManager_icon, represents a semantic entity whose meaning is independent of the platform or the display or the revision of the target application. The semantics of the name is meaningful only in the test. In other words, the test remains portable. If changes in the platform, display, or application require that the exact location of the named object be changed, this is achieved either by editing the object map file or by supplying a different object map file specific for the platform. Synlib provides automated methods to edit or

generate environment-specific object map files. The agent itself does not need any change.

The format of a typical Synlib object map is:

```
! Object Map for the sample program
! Declares the locations named in the test
!   program
Location styleManager_icon 781 51

! Declares the full path of an item named in a
!   focus map
FocusPath StyleManager_Icon_FocusPath
  FrontPanel.ActionIcons.dtstyleIcon
```

Focus Maps

A far superior method of naming GUI objects of interest is to use Synlib's concepts of *focus maps* and *focus paths*. A focus map is a description of the logical organization of the input enabled objects in a widget-based application. An input enabled object is a region in a window that can accept keystrokes or button inputs from a user. Generally these objects are widgets or gadgets used in constructing the user interface.

The method of constructing a focus map is fully described in the Synlib User's Guide.[1] A more complete description of the concept of a focus map and its use in testing X windows applications has been published elsewhere.[2] A focus path is a string of dot-separated names declared in a focus map. For example, StyleManager_Icon_FocusPath is the name of the focus path FrontPanel.ActionIcons.dtstyleIcon, which is a string of dot-separated names declared in the focus map named FrontPanel. Focus maps are described in focus map files. Focus paths, on the other hand, are declared in object map files because, when associated with a window, a focus path identfies an actual object capable of accepting input.

In the example program above, the function SynSelectItem() represents an instruction to the agent to select the object named by the string StyleManager_Icon_FocusPath, which can be declared in the object map file as shown above.

The following is the focus map for the front panel window.

```
!
! Focus map for the default front panel window
!   of the CDE desktop.
!
(FocusMap FrontPanel
 (FocusGroup ActionIcons
    (App_Panel dtpadIcon dtmailIcon dtlockIcon
    dtbeepIcon workspace_One workspace_Three
    workspace_Two workspace_Four exitIcon
    printer_Panel printerIcon dtstyleIcon
    toolboxIcon Help_Panel helpIcon trashIcon
    dtcmIcon dtfileIcon)))
!
```

If the proper focus map and object maps are provided, the agent will apply Synlib embedded rules to decide how to set focus on the named item and then select or activate the item. During execution, Synlib first processes all supplied focus maps and creates an internal representation. Whenever the program refers to a focus path, Synlib decodes the identity of the desired object by analyzing the focus map in which the focus path occurs. Using the declarations in the focus map and applying OSF/Motif supported keyboard traversal specifications, Synlib generates a series of keystrokes to set the keyboard focus to the object indirectly named via the focus path. The rules for transforming the focus path name to the sequence of keystrokes are somewhat complex and have been fully described elsewhere.[2] These rules are embedded in Synlib and are completely transparent to the user.

This example shows the use of a focus map in naming icons in the front panel. Although the example here deals with a simple situation, the same principles and methods can with equal ease be used to name and access objects in deeply embedded structures like menus and submenus. In general, naming objects by means of a focus map is far superior to naming them by means of an object map. Because access to the objects of interest is via a dynamically generated sequence of keystrokes, the programs employing these methods are resistant to changes in window size, fonts, or actual object locations. This makes the tests completely portable across platforms, displays, and other environmental variabilities. Synlib programs using focus maps to name GUI objects need not be changed at all unless the specification of the target application changes.

Using a similar soft coding technique, Synlib makes it possible to create *locale neutral* tests, that is, tests that can verify the behavior of target applications executing in different language environments without undergoing any change themselves. Use of this technique has substantially reduced the cost of testing internationalized GUI applications. A complete description of the concept of locale neutral tests has been published.[4]

Test Execution Architecture

Synlib provides concepts and tools that enable us to create "one test for one application." The tests, assisted by the required environment dependent resource files like object map, focus map, and key map files, can verify the behavior of target applications executing on different platforms, using different displays, and working in very different language environments.

Fig. 2 shows an execution architecture for Synlib tests. A key map file contains declarations to name keystrokes, button events, and sequences of keystrokes and button events. The key map file provides a way to virtualize and name all inputs to be used by a test program. This mechanism is very useful for creating tests for internationalized applications and is fully described in reference 4.

The cost of creating or modifying the environment resource files is minuscule compared to the cost of creating the tests themselves. Thus, the ability to create tests that are insensitive to differences in the execution environment of the target application has been a great productivity boost to our testing efforts.

A feature of Synlib test technology is that it does not require any change in the target application. It does not require that the application code be modified in any way. There is no need to incorporate any test hook in the application, nor is the application required to relink to any foreign test-specific library. Synlib supports a completely noninvasive testing framework. The test is directly executed on the application

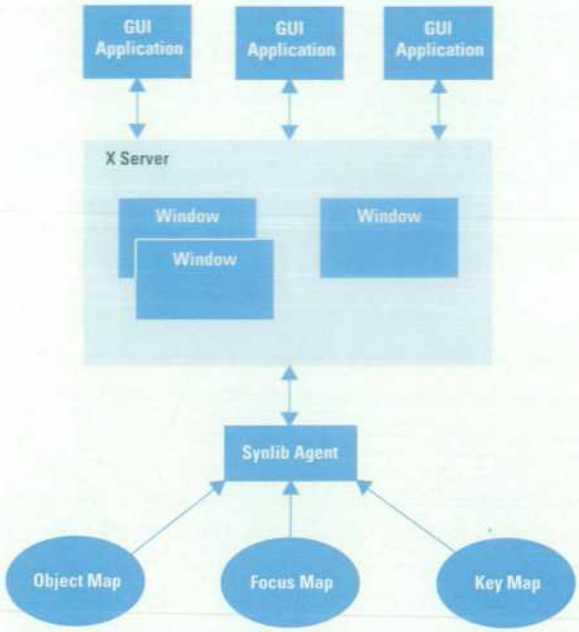


Fig. 2. Synlib test execution architecture.

off the shelf. Synlib even makes it possible to write the tests before the application is ready for testing.[3]

The author originally designed Synlib to solve the problems of GUI testing facing our lab, mainly testing GUI applications that supported a variety of HP displays and operating systems. We designed Synlib to provide a technology that yields robust and platform-insensitive tests at a low cost. Synlib proved to be a marvelous fit for testing the CDE desktop since one of the main conditions was that the tests would have to verify applications running on the platforms of the four participating companies. Essentially it was a problem of creating platform-insensitive tests, a problem that we had already solved. The success of Synlib in this endeavour is shown by the large body of functioning test suites for the complex applications of the CDE desktop.

Acknowledgments

The development of Synlib has benefited from the comments, criticism, and support of many people. The author wishes to thank everyone who willingly came forward to help mature this technology. Harry Phinney, Fred Taft, and Bill Yoder were the first to create test suites for their products using Synlib. Their work proved the value of Synlib to the rest of the laboratory. Subsequently, Bob Miller allowed his group to experiment with Synlib, which led to its adoption as the testing tool for CDE. Thanks Bob, Harry, Fred, and Bill. Julie Skeen and Art Barstow volunteered their time to review the initial design of Synlib. A very special thanks is due Julie. In many ways the pleasant and intuitive user interface of Synlib can be traced to her suggestions. Thanks are also due the engineers at the multimedia lab who proved the effectiveness of Synlib in testing multimedia applications. Ione Crandell empowered this effort. Kristann Orton wrote many of the man pages. Dennis Harms and Paul Ritter effectively supported Synlib in many stormy CDE sessions. Thanks Dennis, Kritann, and Paul. Michael Wilson taught me how Synlib could solve the knotty problem of testing hyperlinked systems. Thanks, Mike. Claudia DeBlau and Kimberly Baker, both of Sun Microsystems, helped in developing Synlib's interface to the Test Environment Toolkit (TET). Finally the author thanks Ken Bronstein. Ken appreciated the value of this unofficial job from the very beginning. Ken's unwavering support has been crucial to the continued development of this technology.

References

1. S.L. Chakrabarti, *Synlib User's Guide—An Experiment in Creating GUI Test Programs*, Hewlett-Packard Company.
2. S.L. Chakrabarti, "Testing X Clients Using Synlib and Focus Maps," *The X Resource*, Issue 13, Winter 1995.
3. S.L. Chakrabarti, R. Pandey, and S. Mohammed, "Writing GUI Specifications in C," *Proceedings of the International Software Quality Conference*, 1995.
4. S.L. Chakrabarti and S. Girdhar, "Testing 'Internationalized' GUI Applications," accepted for publication, *The X Resource*, Winter 1996.

A Hybrid Power Module for a Mobile Communications Telephone

This article describes a 3.5-watt power module designed for a GSM (Global System for Mobile Communications) handheld telephone. The design features proprietary silicon power bipolar devices, lumped elements for input, interstage, and output matching, thick-film alumina ceramic technology, and laser trimmed bias resistors. High-volume manufacturing was a design requirement.

by Melanie M. Daniels

Power modules, as discussed in this article, are the output stage of the RF (radio frequency) amplification chain in a mobile telephone (Fig. 1). Some telephones use integrated circuits as power solutions, but for output power greater than one watt a discrete device is usually used. A power module uses networks to match the discrete stages in a hybrid amplifier.

This article describes a power module designed for a GSM (Global System for Mobile Communications) handheld telephone. GSM telephones transmit in the frequency range of 880 to 915 MHz. The peak transmitter carrier power for power class 4 is 3.5 watts at 1/8 duty cycle. Unlike other TDMA (time division multiple access) systems, it is possible to run a GSM power module close to compression because the amplitude is constant using GMSK (Gaussian minimum phase shift keying) modulation. The pulse width of the transmission burst is 577 microseconds, and the rise time of the power module must be less than 2 µs. It is necessary to supply full output power at a supply voltage of 5.4 volts (five NiCad cells at end of life) with 40% efficiency and 0-dBm input power. This is a requirement of the customer for the phone to be competitive. Future generations of phones may use only four NiCad cells or other battery types and voltages. Of course, a handheld phone must be inexpensive and small and have long talk time (i.e., efficiency) and this dictates the specifications for the power module.

The design goals called for the power module to be small, inexpensive, user friendly, efficient, and manufacturable in volume, and to supply full output power.

Silicon bipolar devices were chosen over GaAs FET devices for this product because of their cost advantages and the fact that HP had developed new silicon power devices that met the stringent requirements of applications in which GaAs had traditionally been used (i.e., low device voltages and excellent efficiency).

Fig. 2 is a photograph of the power module. The schematic diagram, Fig. 3, shows the electrical design of the power module. The bias circuits must be simple and fast because of the pulsed nature of the GSM modulation. Because of the

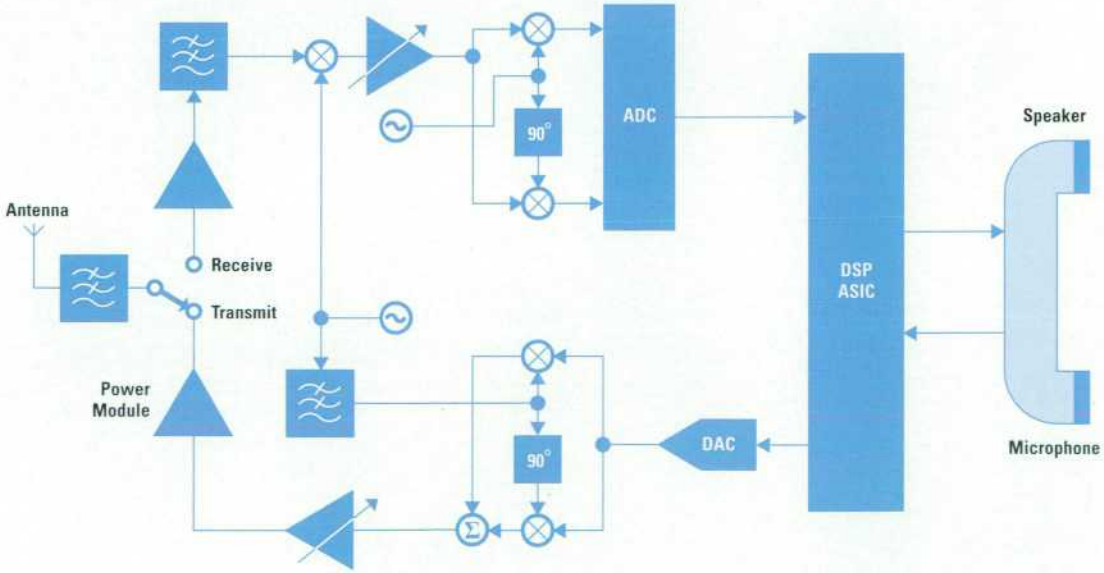


Fig. 1. Block diagram of a typical handheld digital telephone.

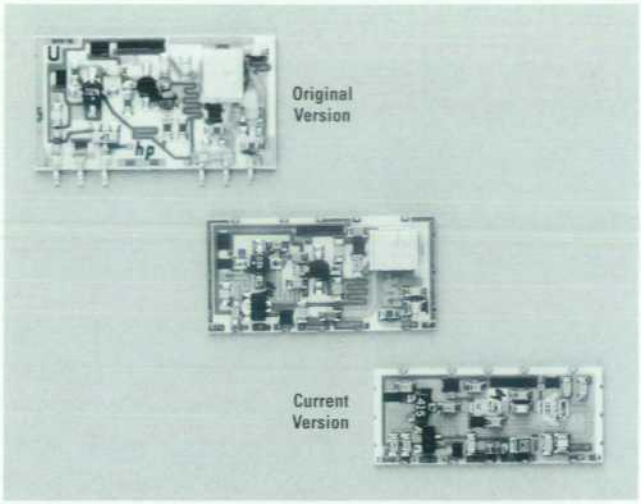


Fig. 2. GSM power module evolution.

low voltage requirements, proprietary silicon power bipolar devices were developed. The collector of each stage simply has an RF choke to the 5.4V minimum supply voltage, V_{cc}, and a bypass capacitor to ground. The base voltage supply is used to turn the amplifier on and off and to control the output power level of the module. The control voltage, V_c, is pulsed at 1/8 duty cycle with a square wave from 0V when the module is off to 4.0V when the module supplies full output power. The base of each stage has a series resistor to the control voltage. This resistor is adjusted to compensate for each transistor's current gain, β. This is done using active laser trimming and will be discussed as a separate topic. Since the power control voltage supplied by the phone does not have the capability of supplying the base current of each stage, a low-frequency n-p-n transistor, Q4, is used to buffer the control voltage. The collector of Q4 is biased by the supply voltage, Vcc. The base of Q4 is driven by the power control voltage and the emitter supplies the necessary voltage and current to the base of each RF stage.

The RF design uses lumped elements for input, interstage, and output matching. The design requires three stages to achieve the gain requirements. The first stage is a driver stage that is class-A biased. The second and third stages are class-AB biased for efficiency.

The third-stage transistor also has some internal matching within the package. The input impedance of the silicon power transistor chip is about 1.5 ohms. This must be transformed up to a larger impedance by a matching network that is physically as close to the chip as possible. This is achieved using a 0.001-inch-diameter bond wire as a series inductor from the base of the chip to a shunt MOS capacitor at the input of the transistor package (Fig. 4). This configuration makes a very high-Q input matching network. The exact value of capacitor and the length of bond wire had to be empirically optimized to achieve the maximum transformation within the transistor package.

The most critical and sensitive part of the matching networks is the output of the final stage. High-Q lumped-element components are used in the output matching network to achieve the low losses necessary to meet the efficiency requirements.

Since the design has more than 45 dB of small-signal gain in a 1-inch-by-0.5-inch package, stability and isolation were quite challenging. The placement and values of the RF chokes and decoupling capacitors were critical. Large-value capacitors could not be placed on the base bias network, since this would slow down the pulse response of the module.

Mechanical Design

As previously mentioned, some of the primary design goals were (1) low cost because this is a commercial product, (2) small size to allow phone manufacturers to design smaller products for portability (also a competitive advantage for HP), and (3) compatibility with high-volume manufacturing. In addition, the power module component had to be supplied as a surface mount component in tape-and-reel form. The mechanical design of the power module turned out to be one of the most challenging parts of the development project. At the time the project was started, most competitors were using soft board for the substrate material and surface mount lumped components for matching. This material definitely

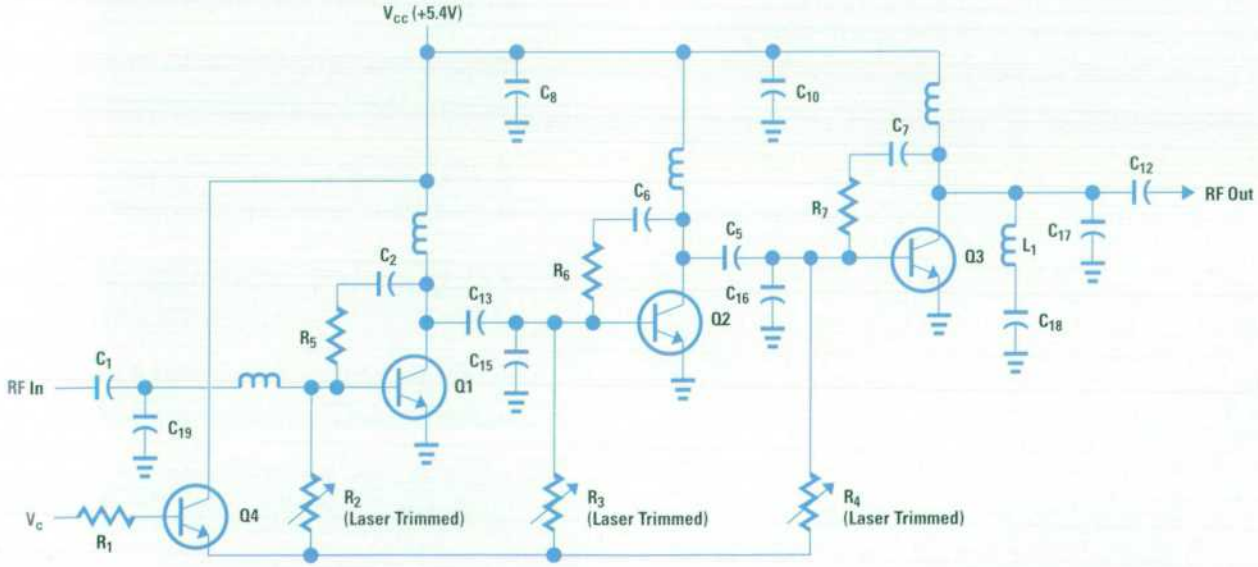


Fig. 3. Schematic diagram of the GSM power module.

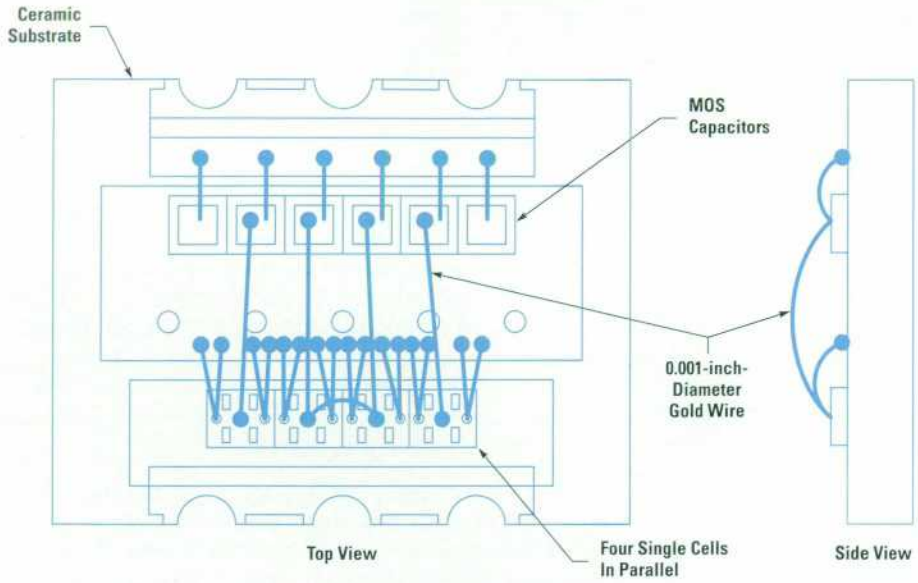


Fig. 4. Packaged output stage transistor.

meets the cost criteria, but there were thermal, RF matching, and laser trimming limitations. Thick-film alumina ceramic technology was chosen for the board material. Even though the material is more expensive, this is offset by the fact that the RF matching networks are more compact because of the high dielectric constant $\varepsilon_r = 9.1$. Also, the resistors and inductors can be printed on the board, thus reducing the part count. Ceramic has superior thermal conductivity compared to soft boards. The most persuasive reason for ceramic substrates is that they do not require a metal carrier to be surface mounted. The vias in a ceramic board can be filled with metal paste so components can be placed directly on top of the via. This reduces the emitter-to-ground inductance for the transistors and gives superior gain and efficiency performance. This factor also reduces the size of the module to 1 inch by 0.4 inch. Standard surface mount components on PdAg traces are used for lumped-element matching and custom surface mount packages are used for the RF transistors.

The inputs and outputs of the power module use wraparound edge vias. This is commonly referred to as an LCC (leadless chip carrier) for surface mount component manufacturers. It is inexpensive because no leadframes need to be attached. The metal thick film used in the vias must pass all solderability tests.

Volume Assembly

Fig. 5 shows the process flow for manufacturing the power modules. Modules are built in array form with 40 modules per 4-inch-by-4-inch array. More modules per array reduces the ceramic substrate cost and the surface mount assembly cost but also increases the complexity of substrate manufacturing. The boards are populated by a subcontractor with standard pick-and-place equipment, then reflowed at a peak temperature of 220°C using SN96 solder paste. The high-temperature reflow was chosen to prevent a secondary reflow by the customer when the power module is surface mounted into the telephone. Developing the reflow profiles with the chosen thick-film paste and high-temperature solder was not a trivial task.

The populated arrays are then actively laser trimmed. Each base bias resistor (three per module) must be trimmed with

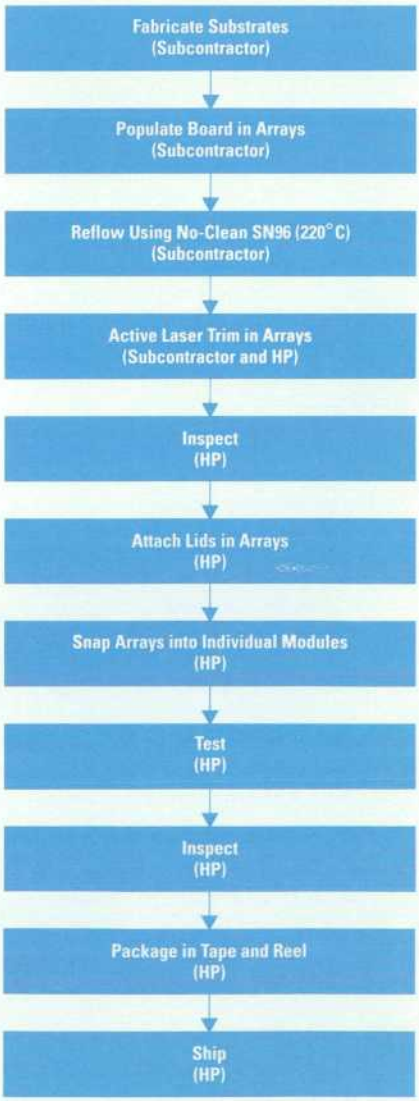


Fig. 5. Volume assembly process flow.

the laser to set the transistor collector current. This is to compensate for β variation in device fabrication. The simpler bias scheme was chosen because we couldn't afford the cost and efficiency loss of an active bias circuit. Developing the laser trim process was another challenging aspect of this product.

Two of the transistors are biased off (have the bases grounded) while the third is being trimmed. This is necessary to avoid oscillations caused by the high gain and the difficulty of adequate grounding while the modules are in array form. Extensive development was required to obtain good RF grounding on the backside of the module and in the bias probes while laser trimming. A grounding gasket made of silver impregnated rubber is used with a vacuum to achieve backside grounding. High-frequency probes with good 50-ohm loads are used on all inputs and outputs to avoid oscillations.

Special software was developed for the laser trimmer. Algorithms were written to compensate for the current drawn through the grounded-base transistors. In addition, the resistors and transistors heat up during the trim process and this has to be compensated. The trim process has to be done in a pulsed bias mode, since the module cannot be run in CW mode without thermal runaway. Finally, the output power cannot reach saturation before the maximum control voltage is reached, since this impacts the customer's power control loop. To resolve this issue, the modules are trimmed at a control voltage of 3.2V maximum.

After laser trimming the lids are attached in array form. The array is then separated into individual units, which are tested using automated test equipment. All of the problems addressed for the laser trimming were also present for the automated test process. The module grounding is absolutely critical to testing RF parameters. Developing test fixtures that could test hundreds of modules per hour, three shifts a day, and still retain a good RF ground was critical to the success of the product. The average test time per module is 20 seconds. The automated test programs are complex because of the number of tests that have to be performed and the fact that they all have to be done in pulsed mode.

Transistor Modeling

Transistor modeling was used to develop linear and nonlinear device models for a single cell of the transistor used in the power module. These building blocks were then used to model the entire power module. The modeling effort included correlating measured device data with the models and modifying the models when necessary. The HP Microwave Design System was used for the linear and nonlinear modeling of the device and the package.

The first step was to use parameter extraction techniques to get a Gummel-Poon SPICE model[1] of the single-cell device. Next, models were developed for the packages.

To make low-voltage bipolar transistors, many new processes were used. These devices have fine geometry to achieve the higher gain necessary for talk-time efficiency. This changed many of the model parameters traditionally used for silicon power transistors.

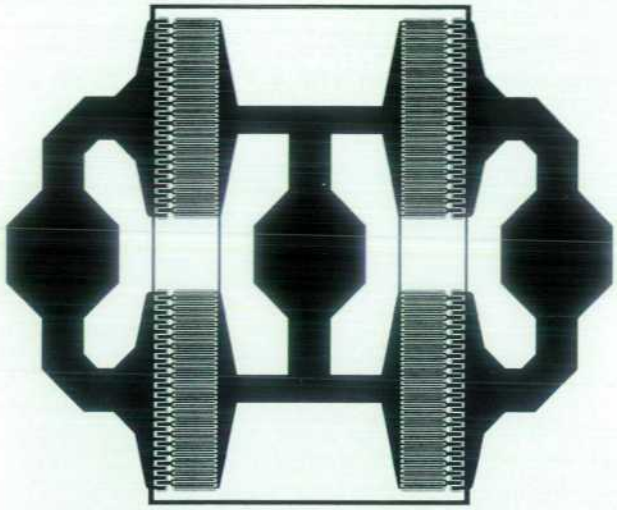


Fig. 6. Single device cell.

The single device chosen for modeling actually consists of four separate quarter cells on each device as shown in Fig. 6. Each quarter cell has 40 emitter fingers with each finger having an approximate capacity of 2 mA of continuous current. With the existing technology it was not possible to extract the parameters of the entire device with 160 fingers and a maximum current of 360 mA, so the quarter cell with 40 fingers was used for the parameter extraction. These devices have HP proprietary epitaxial material and geometries.

The device was placed into the standard 86 package shown in Fig. 7 for the second stage of the power module. This is a

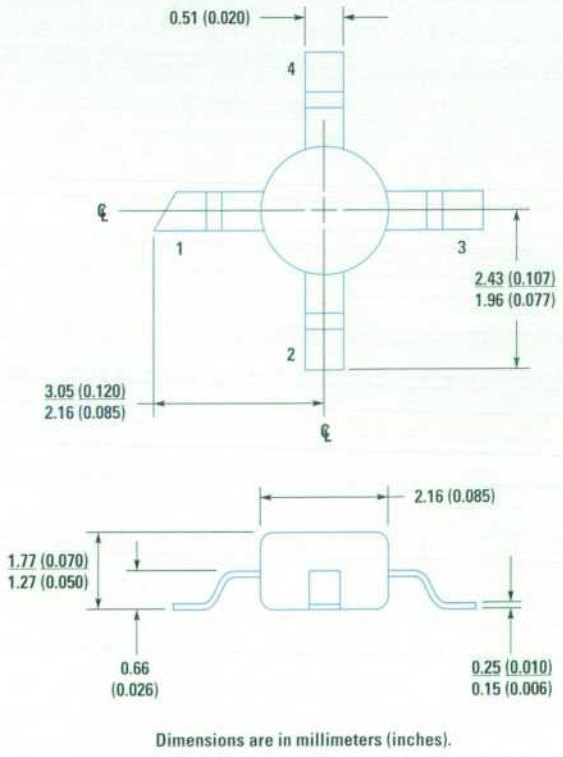


Fig. 7. Standard 86 package.

plastic encapsulated package used for high-volume manufacturing and low power dissipation. The output stage has a custom package as shown in Fig. 4. Thermal dissipation is not a major issue in our application because of the 12.5% duty cycle pulsed operation.

The Ebers-Moll model[2,3] is a good general-purpose model, but is not sufficient over wide regions of operation because of approximations in the equations. The Gummel-Poon model[1] takes second-order effects into account and gives more accurate results. The model we used is a modified Gummel-Poon model.[4] The n-p-n transistor dc and ac circuits are shown in Figs. 8 and 9 respectively.

Among the second-order effects included in the Gummel-Poon model is junction capacitance in the emitter, base, and collector. The Ebers-Moll model holds these capacitances constant but they are more accurately modeled in the Gummel-Poon model as functions of junction voltage. A total of nine model parameters are needed to model the junction capacitance accurately.[5]

Finite resistance and the effects of surfaces, high current injection, and nonideal diodes make the collector and base current vary nonuniformly with base-to-emitter voltage. These effects are modeled using two diodes.

Another complex second-order effect is the variation of unity-gain bandwidth, f_T, with I_c and V_{ce}. At low currents, f_T is almost totally dependent on the g_m of the device and the junction capacitance.[5] At moderate currents, the diffusion capacitance at the base-emitter and base-collector junctions starts to cancel any increase in g_m, which causes f_T to remain constant. This is the constant transit time region. At high currents, the forward transit time of the transistor increases because of space charge limited current flow, which leads to base widening and two-dimensional injection effects.

All of the parameters of the modified Gummel-Poon model are extracted using a modeling system developed by HP. Complete and accurate model parameter sets can be obtained in about two hours using Hewlett-Packard's test system,

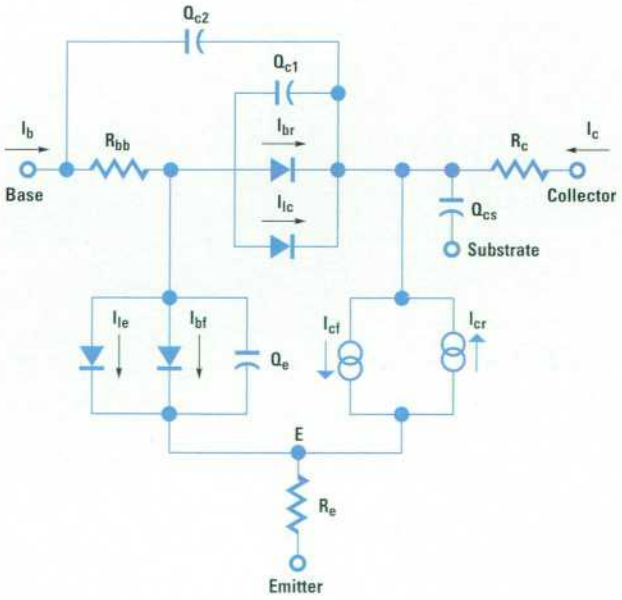


Fig. 8. Modified Gummel-Poon n-p-n dc model.

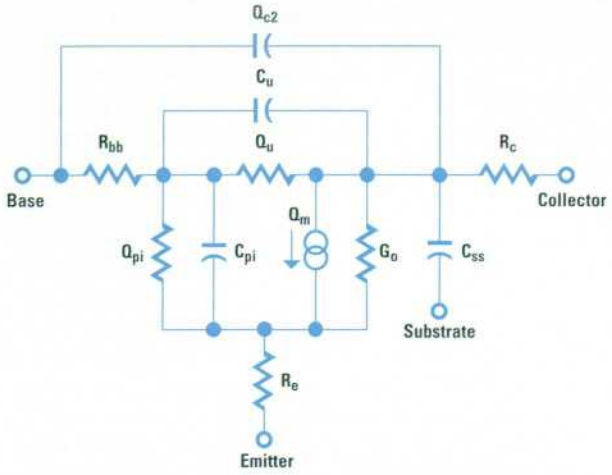


Fig. 9. Modified Gummel-Poon n-p-n ac model.

test and modeling software, and new modeling methods.[6] A through-reflect-line (TRL) calibration method is used and the fixture and bond wires to the device are completely characterized and deembedded to get an accurate model of the device alone.

At the time this work was completed, it was not possible to measure large transistors because of the limitations of the dc power supplies and thermal dissipation problems. To overcome this difficulty, one quarter of the cell was extracted. This was possible since there was a small quarter-cell test transistor on the original engineering wafer as shown in Fig. 10. The Gummel-Poon SPICE file that was obtained by HP's parameter extraction software of this quarter cell is shown in Fig. 11. The schematic for the SPICE file can be seen in Fig. 12.

Since this work was completed, HP has developed a pulsed parameter extraction system that can measure power transistors and a power Gummel-Poon model is being developed to compensate for thermal considerations. With the basic model obtained previously, however, the model for the entire device was developed on the HP Microwave Design System by paralleling four of the quarter cells as shown in Fig. 13. The quarter cell has one base and one emitter pad as shown previously in Fig. 10. The entire cell has one base pad for all

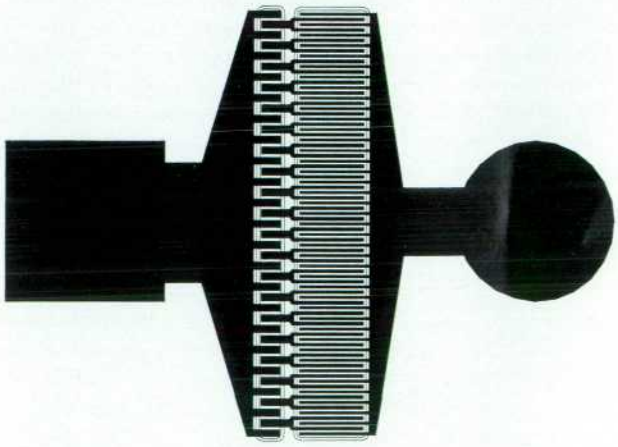


Fig. 10. Quarter device cell.

```
.SUBCKT melanie 1 2 3
LE 3 4 3E-10
LB 2 5 6E-10
CFIXBE 2 3 1E-14
CFIXBC 2 1 3E-14
CFIXEC 1 3 2E-13
CPADBC 5 1 4.1E-13
CPADEC 4 1 4.1E-13
Q1 1 5 4 NPN
+ AREA = 1
.MODEL NPN NPN
+ IS = 3.598E-15
+ BF = 280.7
+ NF = 0.9935
+ VAF = 33.16
+ IKF = 299.9
+ ISE = 9.91E-11
+ NE = 2.399
+ BR = 54.61
+ NR = 0.9886
+ VAR = 1.511
+ IKR = 81
+ ISC = 8.674E-13
+ NC = 1.587
+ RB = 0.752
+ IRB = 0
+ RBM = 0
+ RE = 2.448
+ RC = 1.228
+ XTB = 0
+ EG = 1.11
+ XTI = 3
+ CJE = 5.055E-12
+ VJE = 1.148
+ MJE = 0.5965
+ TF = 1.6E-11
+ XTF = 0.006656
+ VTF = 0.02785
+ ITF = 0.001
+ PTF = 23
+ CJC = 1.352E-12
+ VJC = 0.4776
+ MJC = 0.2508
+ XCJC = 0.001
+ TR = 1E-09
+ FC = 0.999
.ENDS
```

Fig. 11. Quarter-cell SPICE parameter extraction output.

four cells and one emitter pad for two cells as shown in Fig. 6. This must be compensated in the models. The parameter extraction SPICE file (Figs. 11 and 12) clearly shows the capacitance of both the base and emitter pads, CPADBC and CPADEC, respectively. The two-emitter-pad capacitance and the one-base-pad capacitance are added to the model of the four parallel quarter cells (Fig. 13) as components CMP103

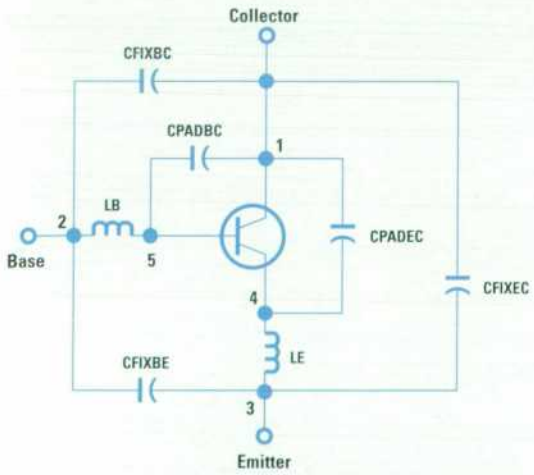


Fig. 12. Quarter device cell schematic of the SPICE file.

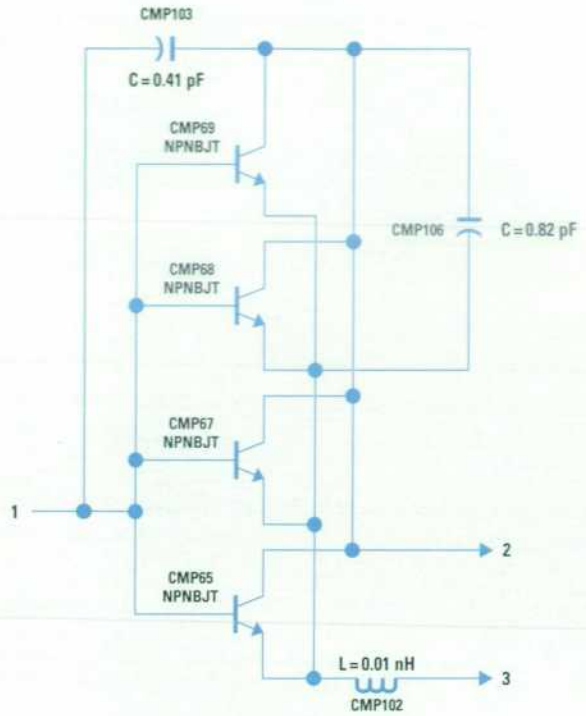


Fig. 13. HP Microwave Design System model for a full single cell.

and CMP106. This file is the basis for the models of the full device.

Separately, similar parameter extraction methods were performed to model the 86 package and the custom output stage package. The 86 package has been modeled extensively for use with other products and can be found in the HP Components Data Book.[7] The HP Microwave Design System package model for the 86 package is shown in Fig. 14.

Both of these models were used for linear and nonlinear designs of the power module. Agreement between the measured performance and the simulated performance was excellent for the linear design (within ±0.5 dB of gain) and good for the nonlinear design (within ±2 dBm of the 1-dB compression point). The performance of the models was good because this was a pulsed application and deviations because of thermal considerations were not a factor.

Results

The GSM power module was engineering released in November 1993 and manufacturing released in May 1994. Over one million of the original GSM power module have been shipped and the module has since gone through two design upgrades to smaller and leadless modules to satisfy customer redesign needs. We now have a manufacturing line and processes that are being used for new products. The module has been a success in the power cellular market.

Project Management and Acknowledgements

This was and is a global project. The customers and sales offices are in Europe and the wafer fab is in Newark, California. The marketing team was originally in San Jose, California, the R&D team was in Folsom, California, and the manufacturing team was in Malaysia and Folsom. We had

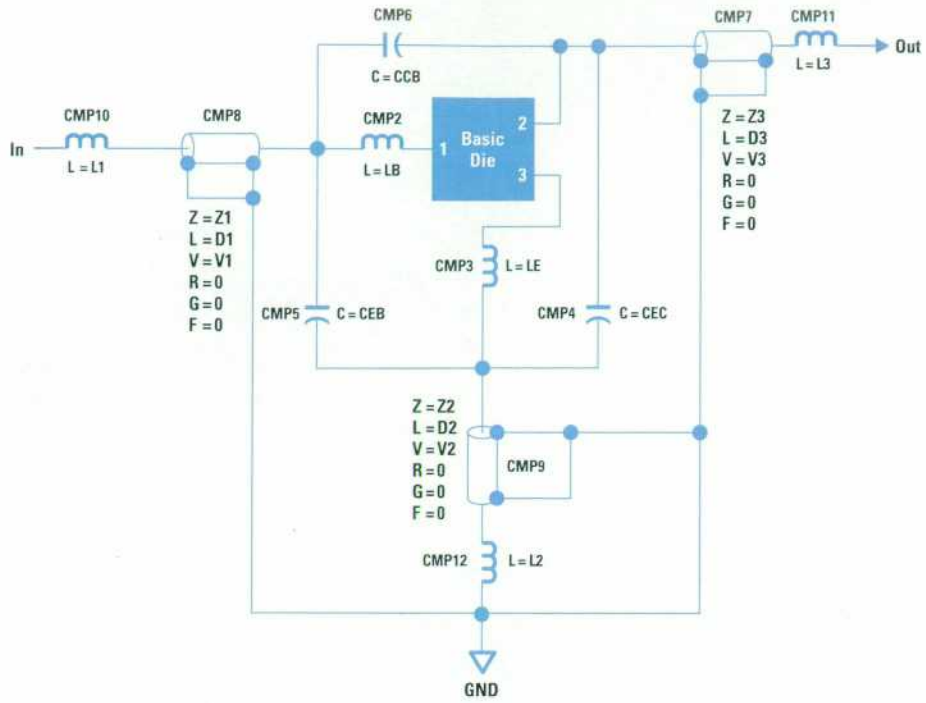


Fig. 14. Simplified HP Microwave Design System model of the second-stage transistor in the 86 package.

never worked together before and there were large learning curves. HP's Malaysia operation had never manufactured a product using thick-film surface mount technologies with the complex RF testing and active laser trimming required. The engineering team had never designed a product for these high volumes. Communication and project management were critical. Many hours were spent on the phone, in meetings, writing email, and traveling. I would like to acknowledge Hanan Zurkowski, Thomas Franke, Rich Levitsky, Mark Dunn, the power module groups in Folsom and Malaysia, and the field sales team for their contributions to the success of this project.

References

1. H.K. Gummel and H.C. Poon, "An Integral Charge Control Model of Bipolar Transistors," *Bell System Technical Journal*, May-June 1970, pp. 827-852.

2. R. Pierret, *Modular Series on Solid State Devices, Volumes I and II*, Addison-Wesley Publishing Company, 1983.

3. I. Getreu, "Modeling the Bipolar Transistor, A New Series," *Electronics*, September 19, 1974, pp. 114-120.

4. *HP 94453A-UCB Bipolar Transistor Model and Parameter Extraction, Section 5*, Hewlett-Packard Company, 1993.

5. I. Getreu, "Modeling the Bipolar Transistor, Part 2," *Electronics*, October 31, 1974, pp. 71-75.

6. M. Dunn, "Device Model and Techniques for Fast Efficient Nonlinear Parameter Extraction of BJT Devices," *The HP-EEsof High-Frequency Modeling and Simulation Seminar*, Hewlett-Packard Company, 1994.

7. *Communication Components Catalog—GaAs and Silicon Products*, Hewlett-Packard Company, 1993.

Bibliography

1. G. Gonzalez, *Microwave Transistor Amplifiers*, Prentice-Hall, 1984.

2. *Simulating Highly Nonlinear Circuits using the HP Microwave and HP RF Design Systems*, Application Note 85150-2, Hewlett-Packard Company, 1993.

3. T.K. Ishii, *Microwave Engineering*, Harcourt Brace Jovanovich, 1989.

4. P.W. Tuinenga, *SPICE, A Guide to Circuit Simulation and Analysis Using PSPICE*, Prentice-Hall, 1992.

Automated C-Terminal Protein Sequence Analysis Using the HP G1009A C-Terminal Protein Sequencing System

The HP G1009A is an automated system for the carboxy-terminal amino acid sequence analysis of protein samples. It detects and sequences through any of the twenty common amino acids. This paper describes a number of applications that demonstrate its capabilities.

by Chad G. Miller and Jerome M. Bailey

Peptide and protein molecules are composed of amino acid molecules covalently coupled end-to-end through peptide bonds, resulting in linear sequences of the amino acid residues. These polypeptide molecules have two chemically distinguishable termini or ends, identified as the amino (N) terminus and the carboxy (C) terminus. The N-terminal and C-terminal amino acid residues define the ends of the amino acid sequence of the polypeptide as depicted in Fig. 1.

The amino acid sequence of the polypeptide molecule imparts its chemical properties, such as the 3D spatial conformation, its biological properties including enzymatic function, its biomolecular recognition (immunologic) properties, and its role in cellular shape and architecture. It is the actual amino acid sequence of the polypeptide (the order in which the amino acid residues occur in the polypeptide from terminus to terminus) rather than the amino acid composition as a random linear arrangement that is crucial in defining the structural and functional attributes of peptides and proteins.

C-terminal amino acid sequence analysis of peptide and protein samples provides crucial structural information for bioscientists. Determination of the amino acid sequence constituting the C-termini of proteins is required to define or verify (validate) the full-length structural integrity of native proteins isolated from natural sources or recombinantly expressed protein products that are genetically engineered.

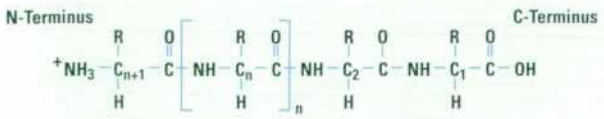


Fig. 1. Amino acid sequence of a polypeptide depicting the N-terminal end (indicated by the subscript n+1) and the C-terminal end (indicated by the subscript 1) of the molecule, with the intervening amino acid residues indicated by the subscript n. C-terminal sequence analysis begins at the C-terminal amino acid residue and sequentially degrades the polypeptide toward the N-terminal amino acid residue.

The cellular processing of proteins at the C-terminus is a relatively common event yielding proteins with varying degrees of amino acid truncations, deletions, and substitutions.[1] The processing of the C-terminal end of the protein results in a family of fragmented protein species terminating with different amino acid residues. The N-terminal amino acid sequence, however, may remain identical for all of these protein species. These *ragged C-terminal* proteins often constitute the family of mature protein forms derived from a single gene expression. Many proteins and peptides are initially biosynthesized as inactive large precursors that are enzymatically cleaved and processed to generate the mature, smaller, bioactive forms. This type of *post-translational* processing also serves to mediate the cellular levels of the biologically active forms of peptides and proteins. Information derived solely from DNA analysis does not permit the prediction of these post-translational proteolytic processing events. The identification of the resultant C-terminal amino acids helps elucidate these cellular biosynthetic processes and control mechanisms.

The polypeptide alpha-amino and alpha-carboxylic acid organic chemical functional groups of the N-terminal and C-terminal amino acid residues, respectively, are sufficiently different to require different chemical methods for the determination of the amino acid sequences at these respective ends of the molecule. N-terminal protein sequence analysis has been refined over the course of four decades and is currently an exceptionally efficient automated chemical analysis method. The chemical (nucleophilic) reactivity of the alpha-amino group of the N-terminal amino acid residue permits a facile chemical sequencing process invoking a cyclical degradative scheme, fundamentally based on the reaction scheme first introduced by P. Edman in 1950.[2] Automation of the N-terminal sequencing chemistry in 1967 resulted in a surge of protein amino acid sequence information available for bioscientists.[3]

The much less chemically reactive alpha-carboxylic acid group of the C-terminal amino acid residue proved to be

Abbreviations for the Common Amino Acids

Amino Acid	Three-Letter Code	Single-Letter Code
Alanine	Ala	A
Asparagine	Asn	N
Aspartic Acid	Asp	D
Arginine	Arg	R
Cysteine	Cys	C
Glycine	Gly	G
Glutamine	Gln	Q
Glutamic Acid	Glu	E
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tyrosine	Tyr	Y
Tryptophan	Trp	W
Valine	Val	V

exceedingly more challenging for the development of an efficient chemical sequencing process. A variety of C-terminal chemical sequencing approaches were explored during the period in which N-terminal protein sequencing using the Edman approach was optimized.[4,5] None of the C-terminal chemical reaction schemes described in the literature proved practical for amino acid sequence analysis across a useful range of molecular weight and structural complexity. Carboxypeptidase enzymes were employed to cleave the C-terminal amino acid residues enzymatically from intact proteins or proteolytic peptides. These carboxypeptidase digests were subjected to chromatographic analysis for the identification of the protein C-terminal amino acid residues. This manual process suffered from the inherent enzymatic selectivity of the carboxypeptidases toward the amino acid residue type and exhibited protein sample-to-sample variability and reaction dependencies. The results frequently yielded ambiguous sequence data. The typical results were inconclusive and provided, more often than not, amino acid compositional information rather than unambiguous sequence information. An alternative approach required the proteolytic digestion of a protein sample (typically with the enzyme trypsin), previously chemically labeled (modified) at the protein C-terminal amino acid residue, and the chromatographic fractionation of the resulting peptides to isolate the labeled C-terminal peptide. The isolated peptide was subjected to N-terminal sequence analysis in an attempt to identify the C-terminal amino acid of the peptide. The limited amount and quality of C-terminal amino acid sequence information derived from these considerably tedious, multistep manual methods appeared to apply best to suitable model test peptides and proteins and offered little generality.

HP Thiohydantoin C-Terminal Sequencing Chemistry

The general applicability of a chemical sequencing scheme for the analysis of the protein C-terminus has only very recently been reported and developed into an automated analytical process.[6] The Hewlett-Packard G1009A C-terminal protein sequencing system, introduced in July 1994, automates a C-terminal chemical sequencing process developed by scientists at the Beckman Research Institute of the City of Hope Medical Center in Duarte, California. The overall chemical reaction scheme is depicted in Fig. 2 and consists of two principal reaction events. The alpha-carboxylic acid group of the C-terminal amino acid residue of the protein is modified to a chemical species that differentiates it from any other constituent amino acid residue in the protein sample. The chemical modification involves the chemical *coupling* of the C-terminal amino acid residue with the sequencing coupling reagent. The modified C-terminal amino acid residue is in a chemical form that permits its facile chemical *cleavage* from the rest of the protein molecule. The cleavage reaction generates the uniquely modified C-terminal amino acid residue as a thiohydantoin-amino acid (TH-aa) derivative, which is detected and identified using HPLC (high-performance liquid chromatography) analysis. The remaining component of the cleavage reaction is the protein shortened by one amino acid residue (the C-terminal amino acid) and is subjected to the next round of this coupling/cleavage cycle. The overall sequencing process is thus a sequential chemical degradation of the protein yielding successive amino acid residues from the protein C-terminus that are detected and identified by HPLC analysis.

As shown in Fig. 3, the coupling reaction event of the sequencing cycle begins with the activation of the carboxylic acid function of the C-terminal amino acid residue, promoting its reaction with the coupling reagent, diphenyl phosphoryl-isothiocyanate (DPPITC). The reaction is accelerated in the presence of pyridine and suggests the formation of a peptide phosphoryl anhydride species as a plausible chemical reaction intermediate. The final product of the coupling reaction is the peptide thiohydantoin, formed by chemical cyclization of the intermediate peptide isothiocyanate.

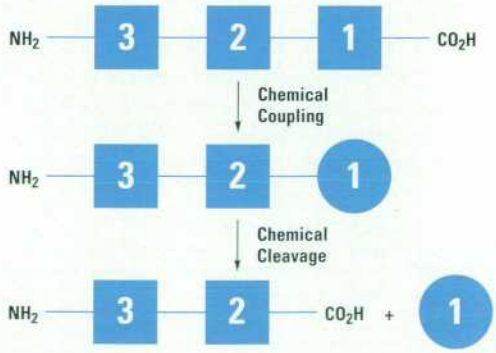


Fig. 2. Overall reaction scheme of the C-terminal sequencing process depicting the initial chemical coupling reaction, modifying the C-terminal amino acid (as indicated by a circle), and the chemical cleavage reaction, generating the C-terminal amino acid thiohydantoin derivative (indicated as a circle) and the shortened peptide.

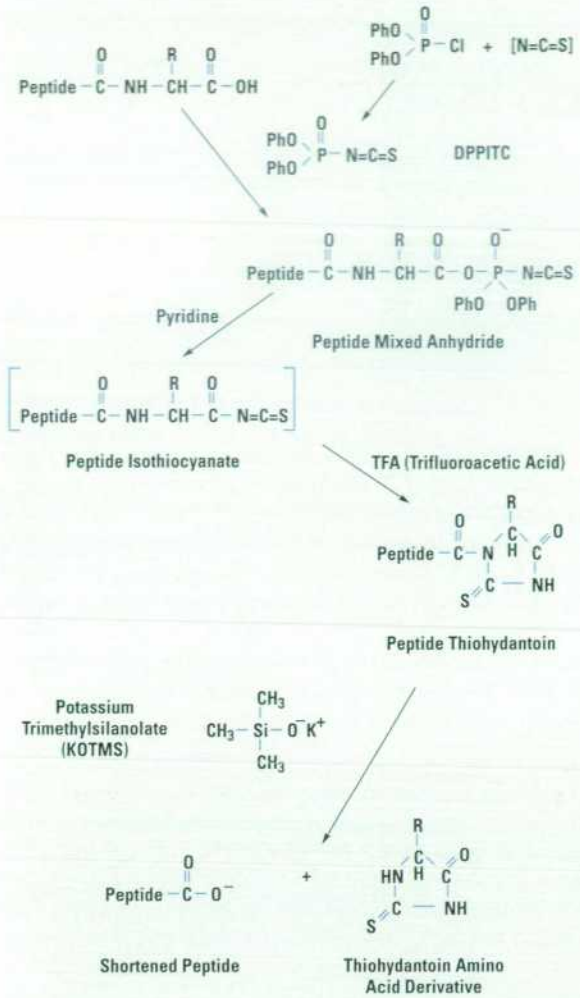


Fig. 3. Detailed reaction scheme of the HP thiohydantoin C-terminal sequencing chemistry. The chemical coupling reactions with diphenyl phosphorylisothiocyanate (DPPITC) generate a mixed anhydride followed by the extrusion of phosphate with ring formation to yield the peptide thiohydantoin. The subsequent chemical cleavage reaction with potassium trimethylsilanolate (KOTMS) releases the C-terminal amino acid thiohydantoin derivative (TH-aa) from the shortened peptide.

The peptide thiohydantoin, bearing the chemically modified C-terminal amino acid, is cleaved with trimethylsilanolate (a strong nucleophilic base) to release the C-terminal thiohydantoin-amino acid (TH-aa) derivative for subsequent analysis and the shortened peptide for the next cycle of chemical sequencing. The thiohydantoin derivative of the C-terminal amino acid residue is chromatographically detected and identified by HPLC analysis at a UV wavelength of 269 nm.

Data Analysis and Interpretation of Results

The data analysis relies on the interpretation of HPLC chromatographic data for the detection and identification of thiohydantoin-amino acid derivatives. The sequencing system uses the HP ChemStation software for automatic data processing and report generation. By convention, the amino acid sequence is determined from the successive single amino acid residue assignments made for each sequencer cycle of analysis beginning with the C-terminal residue (the first sequencer cycle of analysis). The thiohydantoin-amino acid derivative at any given sequencer cycle is assigned by visually examining the succeeding cycle (n+1) and preceding cycle (n–1) chromatograms with respect to the cycle in question (n). The comparison of peak heights (or widths) across three such cycles typically enables a particular thiohydantoin-amino acid derivative to be detected quantitatively, rising above a background level present in the preceding cycle, maximizing in absorbance in the cycle under examination, and decreasing in absorbance in the succeeding cycle. Exceptions to this most basic algorithm are seen in sequencing cycles in which there are two or more identical amino acid residues in consecutive cycles and for the first sequencer cycle, which has no preceding cycle of analysis.

An HPLC chromatographic reference standard consisting of the twenty common amino acid thiohydantoin synthetic standards defines the chromatographic retention time (elution time) for each particular thiohydantoin-amino acid derivative. The TH-aa derivatives chromatographically detected in a sequencer cycle are identified by comparison of their chromatographic retention times with the retention times of the 20-component TH-aa derivatives of a reference standard mixture. The practical assignment of amino acid residues in sequencing cycles is contingent on an increase in peak absorbance above a background level followed by an absorbance decrease for a chromatographic peak that exhibits the same chromatographic retention time as one of the twenty thiohydantoin-amino acid reference standards. A highly robust chromatographic system is required for this mode of analysis to be reliable and reproducible.

The chromatographic analysis of the online thiohydantoin-amino acid standard mixture is shown in Fig. 4. The peaks are designated by the single-letter codes for each of the 20 common amino acid residues (see page 74) and correspond to approximately 100 picomoles of each thiohydantoin derivative. The thiohydantoin standard mixture is composed of the sequencing products of each of the amino acids so that Ser and Cys amino acid derivatives each result in the same

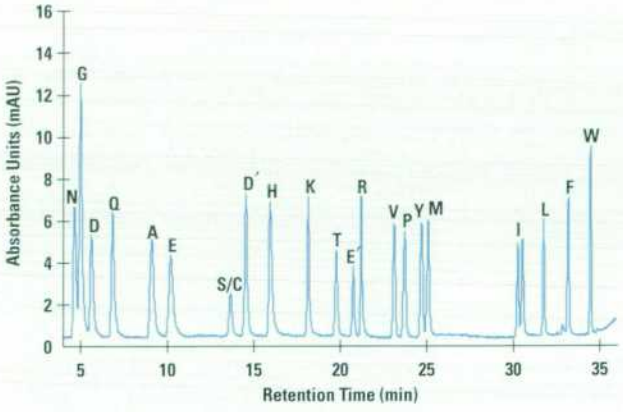


Fig. 4. HPLC chromatogram of the thiohydantoin-amino acid standard mixture (TH-Std). The thiohydantoin-amino acid derivatives of the common amino acids are indicated by their single-letter code designations (see page 74). The peak identified as S/C represents the thiohydantoin derivative of serine and cysteine. The peak designated D' represents the methyl ester sequencing product of aspartic acid and the E' peak represents the methyl ester sequencing product of glutamate. Isoleucine, designated I, elutes as two chromatographically resolved stereoisomeric thiohydantoin derivatives.

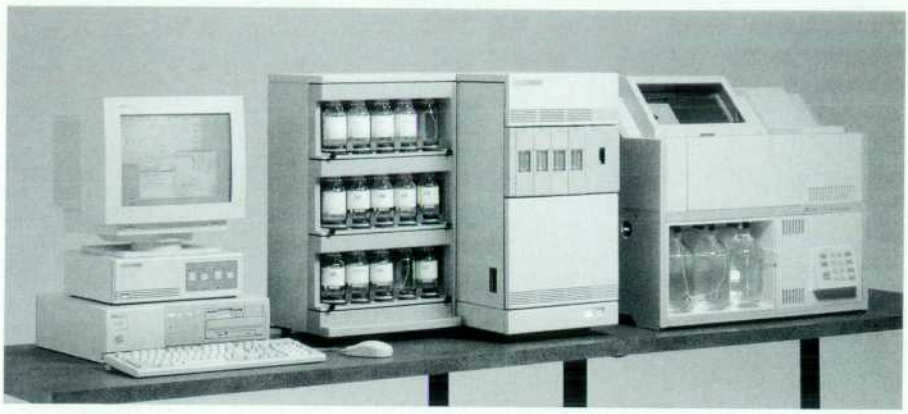


Fig. 5. The HP G1009A C-terminal protein sequencing system consists of the HP C-terminal sequencer, the HP 1090M HPLC, and an HP Vectra computer.

compound, designated S/C, which is the dehydroalanine thiohydantoin-amino acid derivative. The peaks labeled D′ and E′ signify the methyl esters of the thiohydantoin-amino acid derivatives of Asp and Glu.

The C-terminal protein sequencing system is configured with the HP G1009A protein sequencer which automates the chemical sequencing, the HP 1090M HPLC for the chromatographic detection and identification of the thiohydantoin-amino acids, and an HP Vectra personal computer for instrument control and data processing as shown in Fig. 5. The chemical sequencer consists of assemblies of chemically resistant, electrically actuated diaphragm valves connected through a fluid path of inert tubing that permit the precise delivery of chemical reagents and solvents. The fluid delivery system is based on a timed pressurization of reagent and solvent bottles that directs the fluid flow through delivery valves (including valve manifolds) into the fluid path. The sequencer control software functions on a Microsoft® Windows platform and features an extensive user-interactive graphical interface for instrument control and instrument method editing. The sequencer data analysis software is a modified version of the HP ChemStation software with features specific for the analysis and data reporting of the thiohydantoin-amino acid residues.

Sample Application to Zitex Membranes

The sequencing chemistry occurs on Zitex reaction membranes (inert porous membranes), which are housed in Kel-F (inert perfluorinated plastic) sequencer reaction columns. The Zitex membrane serves as the sequencing support for the coupling and cleavage reactions. The membrane is chemically inert to the sequencing reagents and retains the sample through multipoint hydrophobic interactions during the sequencing cycle. The chemical methodology performed on the Zitex membrane enables the sequence analysis of proteins and low-molecular-weight peptide samples. The sequencer sample reaction columns are installed in any one of four available sequencing sample positions, which serve as temperature-controlled reaction chambers. In this fashion, four different samples can be independently programmed for automated sequence analysis.

C-terminal sequence analysis is readily accomplished by directly applying the protein or peptide sample to a Zitex reaction membrane as diagrammed in Fig. 6. The process does not require any presequencing attachment or coupling chemistries. The basic, acidic, or hydroxylic side-chain amino acid residues are not necessarily subjected to any presequencing chemical modifications or derivatizations. Consequently, there are no chemically related sequencing ambiguities or chemical inefficiencies. Protein samples that are isolated using routine separation procedures involving various buffer systems, salts, and detergents, as well as samples that are prepared as product formulations, can be directly analyzed using this technique. The chemical method is universal for any of the 20 common amino acid residues and yields thiohydantoin derivatives of serine, cysteine, and threonine—all of which frequently appear in protein C-terminal sequences.

To be successful, the sequence analysis must provide unambiguous and definitive amino acid residue assignments at cycle 1, since all protein forms—whether they result from internal processing, clippings, or single-residue truncations—are available for analysis at that cycle in their relative proportions.

New Sequencing Applications

The Hewlett-Packard G1009A C-terminal protein sequencing system is capable of performing an expanded scope of sequencing applications for the analysis of peptide and protein samples prepared using an array of isolation and purification techniques. The recently introduced version 2.0 of the HP thiohydantoin sequencing chemistry for the HP G1009A C-terminal sequencer now supports both the "high-sensitivity" sequence analysis of protein samples recovered in 50-to-100-picomole amounts and an increase in the number of

1. Zitex membrane is treated with alcohol.
2. Sample solution is directly applied to membrane and allowed to dry.
3. Membrane is inserted into a sequencer column.

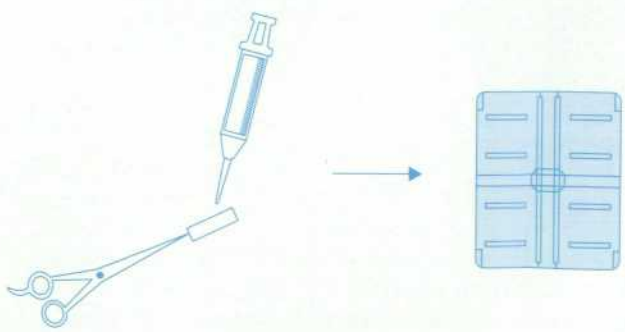


Fig. 6. Procedure for sample application (loading) onto a Zitex C-terminal sequencing membrane.

sequenceable cycles. The high-sensitivity C-terminal sequence analysis is compatible with a great variety of samples encountered at these amount levels in research laboratories.

C-terminal sequence analysis of protein samples recovered from SDS (sodium dodecylsulfate) gel electrophoresis is an important application enabled by the high-sensitivity sequencing chemistry. SDS gel electrophoresis is a routine analytical technique based on the electrophoretic mobility of proteins in gel matrices. The technique provides information on the degree of overall sample heterogeneity by allowing individual protein species in a sample to be resolved. The sequence analysis is performed on the gel-resolved proteins after the physical transfer of the protein components to an inert membrane, such as Teflon, using an electrophoretic process known as electroblotting. SDS gel electrophoresis of native and expressed proteins frequently exhibits multiple protein bands indicating sample heterogeneity or internal protein processing—both of critical concern for protein characterization and production.

The combined capabilities of high-sensitivity sequence analysis and sequencing from SDS gel electroblots has enabled the development of tandem N-terminal and C-terminal sequence analyses on single samples using the HP G1005A N-terminal sequencer in combination with the HP G1009A C-terminal sequencer. This procedure unequivocally defines the protein N-terminal and C-terminal amino acid residues and the primary structural integrity at both termini of a single sample, thereby eliminating multiple analytical methods and any ambiguities resulting from sample-to-sample variability.

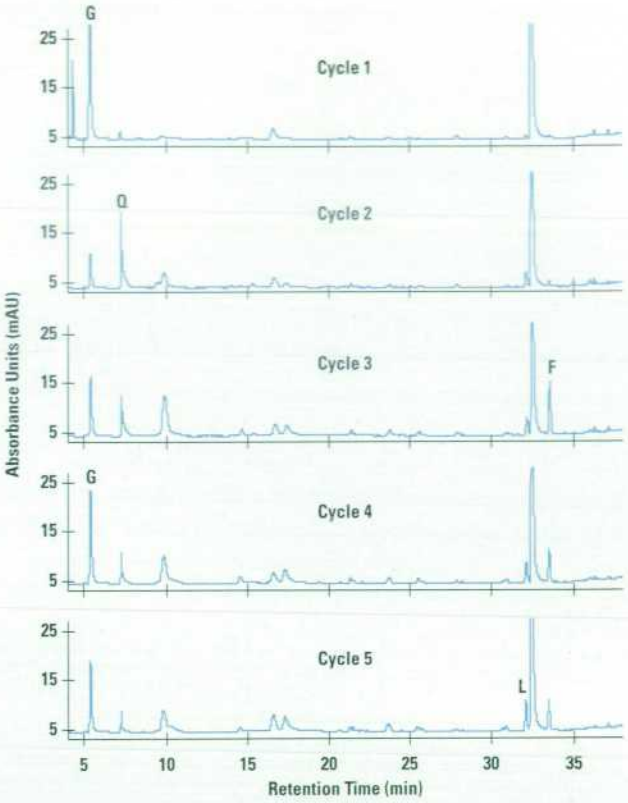


Fig. 7. Cycles 1 to 5 of C-terminal sequencing of 1 nanomole of horse apomyoglobin.

C-Terminal Sequence Analysis Examples

The first five cycles of an automated C-terminal sequence analysis of horse heart apomyoglobin (1 nanomole) are shown in Fig. 7. The unambiguous result observed for cycle 1 confirms the nearly complete homogeneity of the sample, since no significant additional thiohydantoin derivatives can be assigned at that cycle. The analysis continues clearly for five cycles enabling the amino acids to be assigned with high confidence. The results of a sequence analysis of 500 picomoles of recombinant interferon are shown in Fig. 8. The first five cycles show the unambiguous sequencing residue assignments for the C-terminal amino acid sequence of the protein product.

The results of a C-terminal sequence analysis of 1 nanomole, 100 picomoles, and 50 picomoles of bovine beta-lactoglobulin A are shown in Fig. 9 as the cycle-1 HPLC chromatograms obtained for each of the respective sample amounts. The recovery for the first cycle of sequencing is approximately 40% to 50% (based on the amounts applied to the Zitex membranes) and an approximately linear recovery is observed across the range of sample amount. The linear response in the detection of the thiohydantoin-amino acid sequencing products and a sufficiently quiet chemical background permit the high-sensitivity C-terminal sequence analysis.

The automated C-terminal sequencing of the HP G1009A C-terminal sequencer facilitated the detection and identification of a novel C-terminal modification observed for a class of recombinant protein molecules expressed in E. coli that has compelling biological implications. Dr. Richard Simpson and colleagues at the Ludwig Institute for Cancer Research

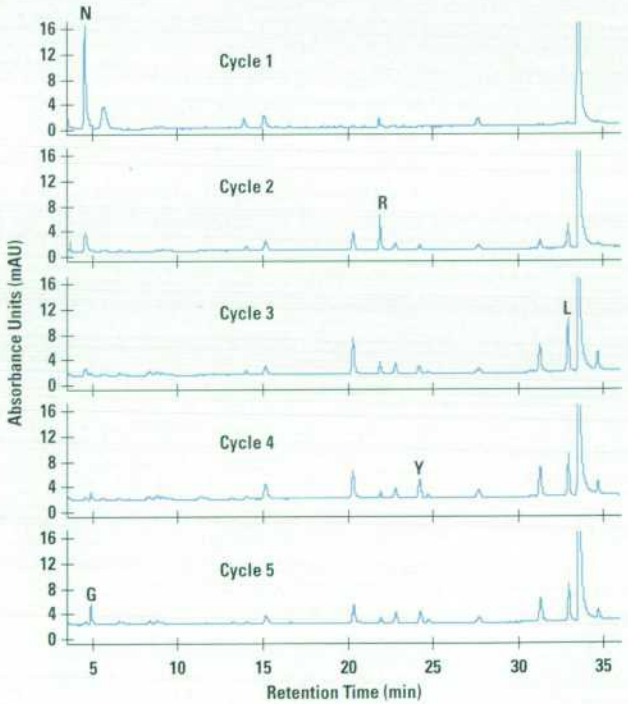


Fig. 8. Cycles 1 to 5 of C-terminal sequencing of 500 picomoles of recombinant beta-interferon applied to Zitex (230 picomoles of Asn recovered in cycle 1).

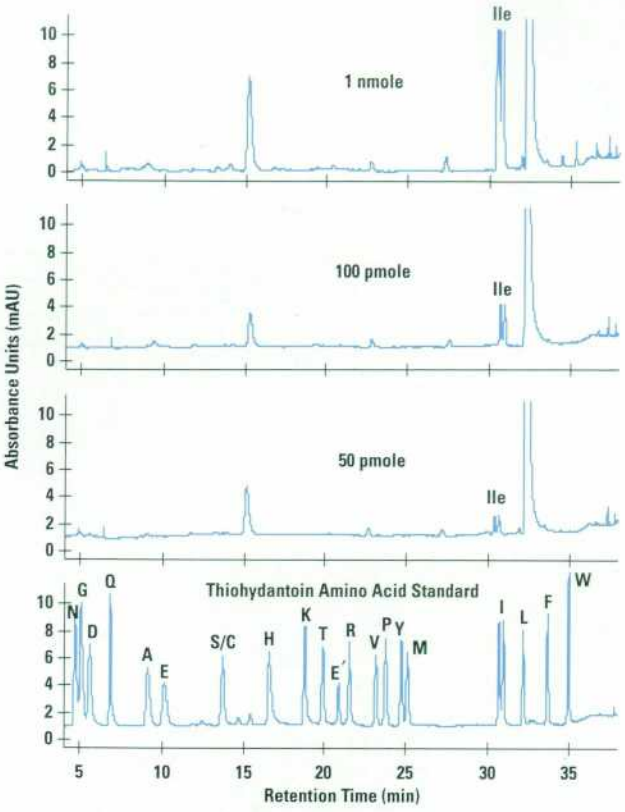


Fig. 9. C-terminal sequence analysis of bovine beta-lactoglobulin A across a 20-fold range of sample amount.

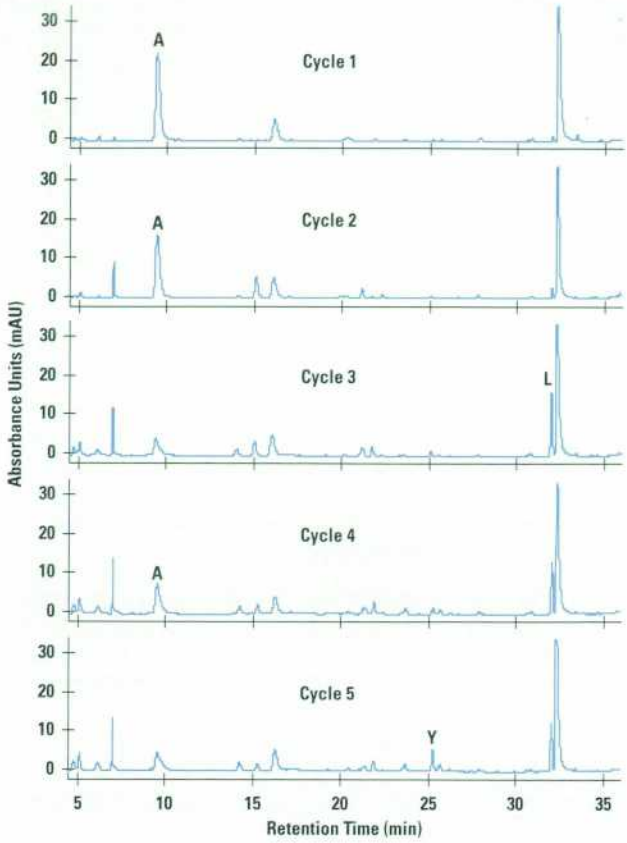


Fig. 10. Cycles 1 to 5 of C-terminal sequencing of recombinant murine interleukin-6 identifying the C-terminal extension peptide (AALAY) as described in the text. (1 nanomole applied to Zitex: 555 picomoles of Ala recovered in cycle 1.)

in Melbourne, Australia reported the identification of a population of C-terminal-truncated forms of murine interleukin-6 molecules recombinantly expressed in E. coli that bear a C-terminal peptide extension.[7] Fig. 10 shows the results of the first five cycles of a C-terminal sequence analysis of a purified form of C-terminal-processed interleukin-6 identifying the amino acid sequence of the C-terminal-appended peptide as Ala(1)-Ala(2)-Leu(3)-Ala(4)-Tyr(5).

High-Sensitivity C-Terminal Sequence Analysis

Protein samples in amounts of 50 picomoles or more are applied in a single step as liquid aliquots to isopropanol-treated Zitex reaction membranes. The samples disperse over the membrane surface where they are readily absorbed and immobilized. The dry sample membranes do not usually require washing once the sample is applied even in those cases where buffer components and salts are present. As a rule, the matrix components do not interfere with the automated chemistry and are eliminated during the initial steps of the sequencing cycle.

Fig. 11 shows the unambiguous residue assignments for the first three cycles of sequence analysis of a 50-picomole sample of bovine beta-lactoglobulin A applied to a Zitex membrane. Again, the linear response in the detection of the thiohydantoin-amino acid sequencing products coupled with a sufficiently quiet chemical background enable high-sensitivity C-terminal sequence analysis. The sequence analysis of 50 picomoles of human serum albumin is shown in Fig. 12. The chromatograms permit clear residue assignments for the first three cycles of sequencing.

C-Terminal Sequence Analysis of Electroblotted SDS Gel Samples

Analytical gels are routinely used for the analysis of protein samples and recombinant products to assess sample homogeneity and the occurrence of related forms. The common observation of several closely associated protein SDS gel bands is an indication either of cellular processing events or of fragmentations induced by the isolation and purification protocols. Because of its capacity to perform high-sensitivity analysis, C-terminal sequence analysis provides a direct characterization of these various protein species and facilitates the examination of their origin and control.

In collaboration with scientists at Glaxo Wellcome Research Laboratories, Research Triangle Park, North Carolina, SDS gel electroblotting procedures have been developed and applied to protein samples of diverse nature and origin.[8] Typically, 50-picomole (1-to-10-microgram) sample amounts are loaded into individual SDS gel lanes and separated according to molecular weight by an applied voltage. Following electrophoretic resolution of the protein samples, the protein bands on the SDS gel are electroblotted to a Teflon membrane. The electroblotted proteins, visualized on the Teflon membrane after dye-staining procedures, are excised in the proper dimensions for insertion directly into C-terminal sequencer columns as either single bands (1 cm) or multiple bands. The visualization stain does not interfere with the

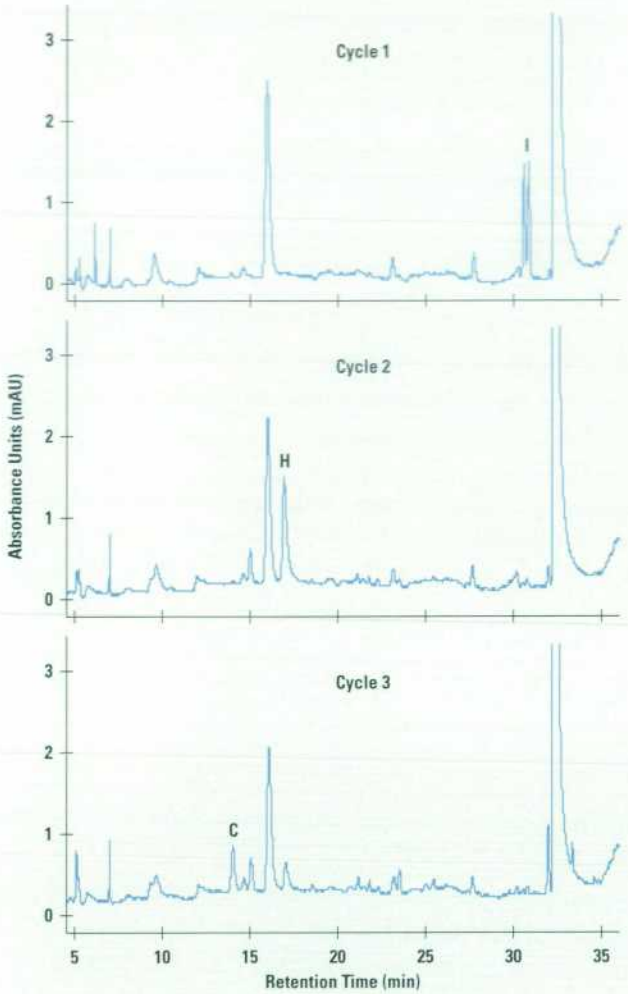


Fig. 11. High-sensitivity C-terminal sequencing of 50 picomoles of bovine beta-lactoglobulin A indicating approximately 50% recovery (24 picomoles) of cycle-1 isoleucine.

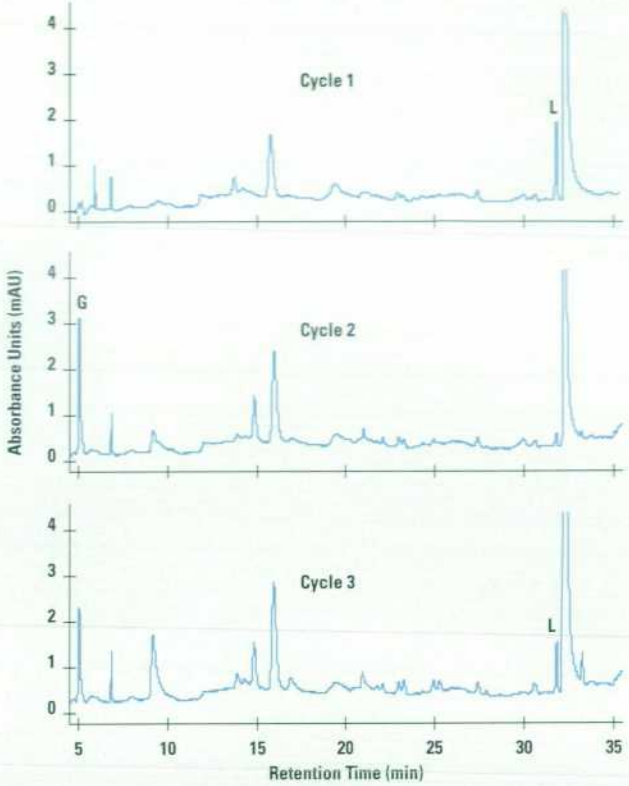


Fig. 12. High-sensitivity C-terminal sequencing of 50 picomoles of human serum albumin indicating approximately 56% recovery (28 picomoles) of cycle-1 leucine.

sequence analysis and the excised bands do not have to be treated with any extensive wash protocol before sequencing.

Fig. 13 shows the results of sequence analysis of approximately 250 picomoles of a bovine serum albumin (68 kDa†) sample loaded into one lane of the SDS gel, subjected to electrophoresis, and electroblotted to a Teflon membrane. The chromatograms for cycles 1 to 3 of the single 1-cm protein band sequenced from the Teflon electroblot illustrate the high sensitivity of the sequencing technique. In Fig. 14, approximately 50 picomoles of a phosphodiesterase (43 kDa) sample were applied to an SDS gel lane, subjected to electrophoresis, electroblotted to Teflon, and excised as a single 1-cm sulforhodamine-B-stained band. C-terminal sequence analysis of the Teflon blot enabled the determination of the extent of C-terminal processing as demonstrated by the presence in cycle 1 of the Ser expected from the full-length form, in addition to the His residue arising from a protein form exhibiting the truncation of the C-terminal amino acid, in about a 10:1 ratio. The expected full-length C-terminal sequence continues in cycle 2 (His) and cycle 3 (Gly), with the truncated form yielding Gly in cycle 2 and Glu in cycle 3. Additional processed protein forms also are identifiable.

† kDa = kilodaltons. A dalton is a unit of mass measurement approximately equal to 1.66×10^{-24} gram.

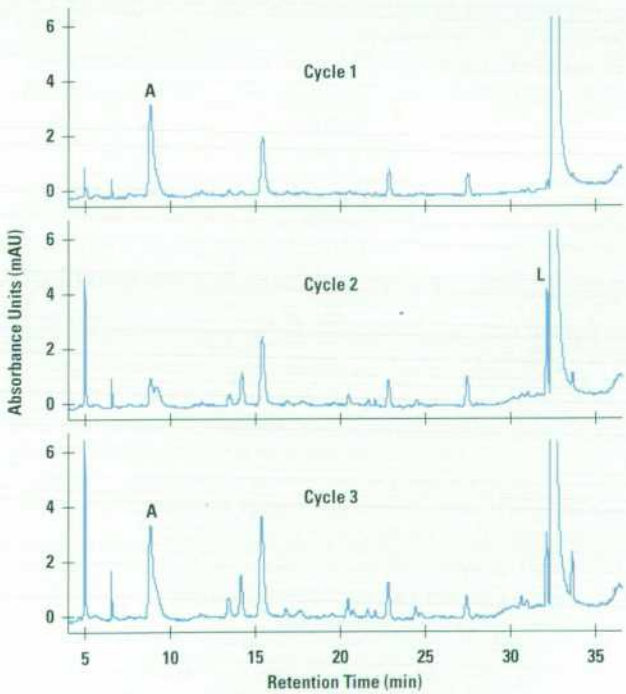


Fig. 13. C-terminal sequence analysis of 250 picomoles of bovine serum albumin applied to an SDS gel, electroblotted to Teflon tape, and subjected to three cycles of analysis. 70 picomoles of Ala recovered in cycle 1.

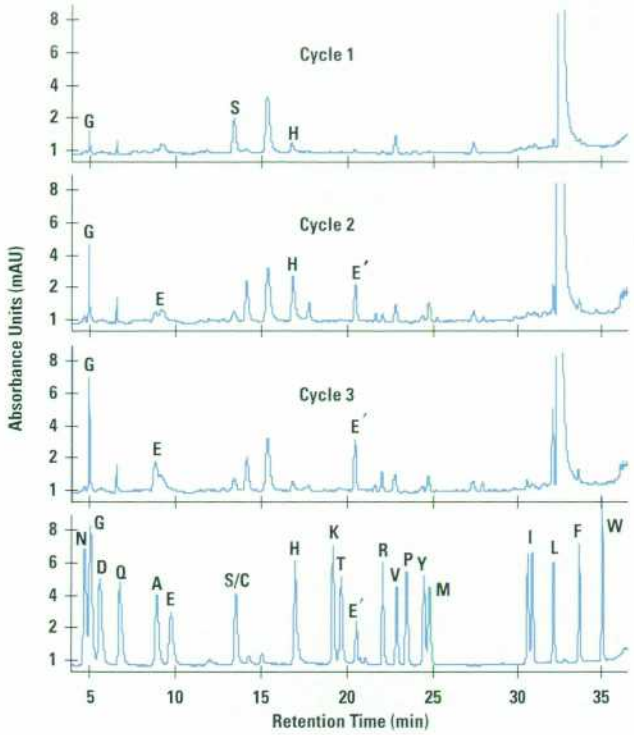


Fig. 14. Cycles 1 to 3 of a C-terminal sequence analysis of 50 picomoles of phosphodiesterase electroblotted from an SDS gel to Teflon tape. Three protein species are identified at cycle 1 of the sequence analysis indicating protein C-terminal processing of the sample.

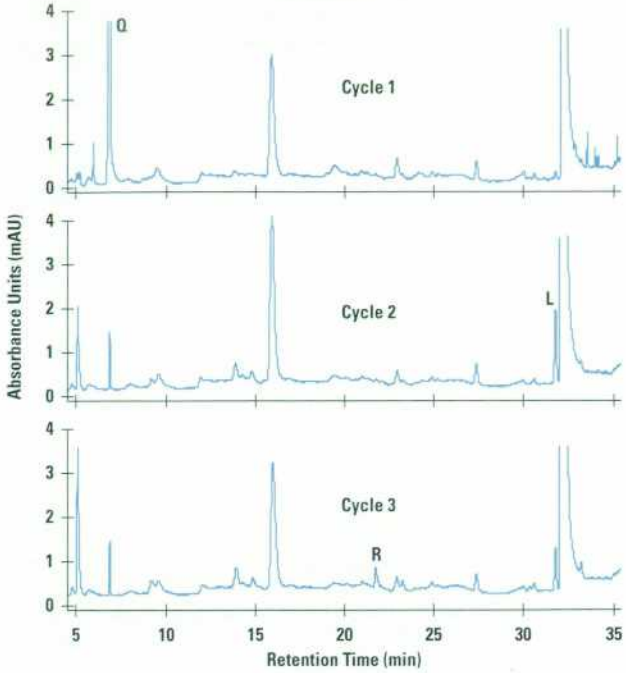


Fig. 15. Cycles 1 to 3 of a C-terminal sequence analysis of polio viral capsid protein, VP2, electroblotted from an SDS gel to Teflon tape.

This application was exploited by researchers at the Department of Biological Chemistry of the University of California at Irvine to investigate the C-terminal amino acid sequences of polio virus capsid proteins and examine additional viral systems.[9] Polio viral capsid proteins VP2 and VP3 were electroblotted to Teflon tape from SDS gels for C-terminal sequence analysis. Figs. 15 and 16 show the results of the sequence analysis, which unambiguously identifies the C-terminal residue in both proteins as Gln, with additional amino acid residue assignments being Gln(1)-Leu(2)-Arg(3) for VP2 and Gln(1)-Ala(2)-Leu(3) for VP3 subunits, respectively.

Tandem N-Terminal and C-Terminal Sequence Analysis

The amino acid sequence analysis of both the N-terminus and the C-terminus of single protein samples enables amino acid sequence identification at the respective terminal ends of the protein molecule in addition to a rapid and reliable detection of ragged protein termini. The tandem sequence analysis process of N-terminal protein sequencing followed by C-terminal protein sequencing of the same single sample provides a combined, unequivocal technique for the determination of the structural integrity of expressed proteins and biologically processed precursors. The analyses are performed on a single sample of protein, thereby eliminating any ambiguities that might be attributed to sample-to-sample variability and sample preparation.

The protein sample is either applied as a liquid volume onto a Zitex membrane or electroblotted to Teflon from an SDS gel and inserted into a membrane-compatible N-terminal sequencer column. The sample is subjected to automated N-terminal sequencing using the HP G1005A N-terminal

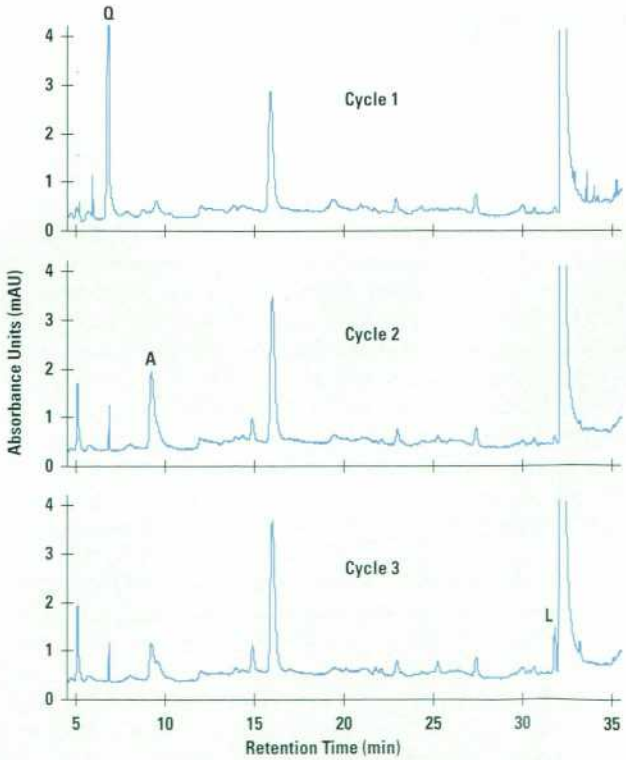


Fig. 16. Cycles 1 to 3 of a C-terminal sequence analysis of polio viral capsid protein, VP3, electroblotted from an SDS gel to Teflon tape.

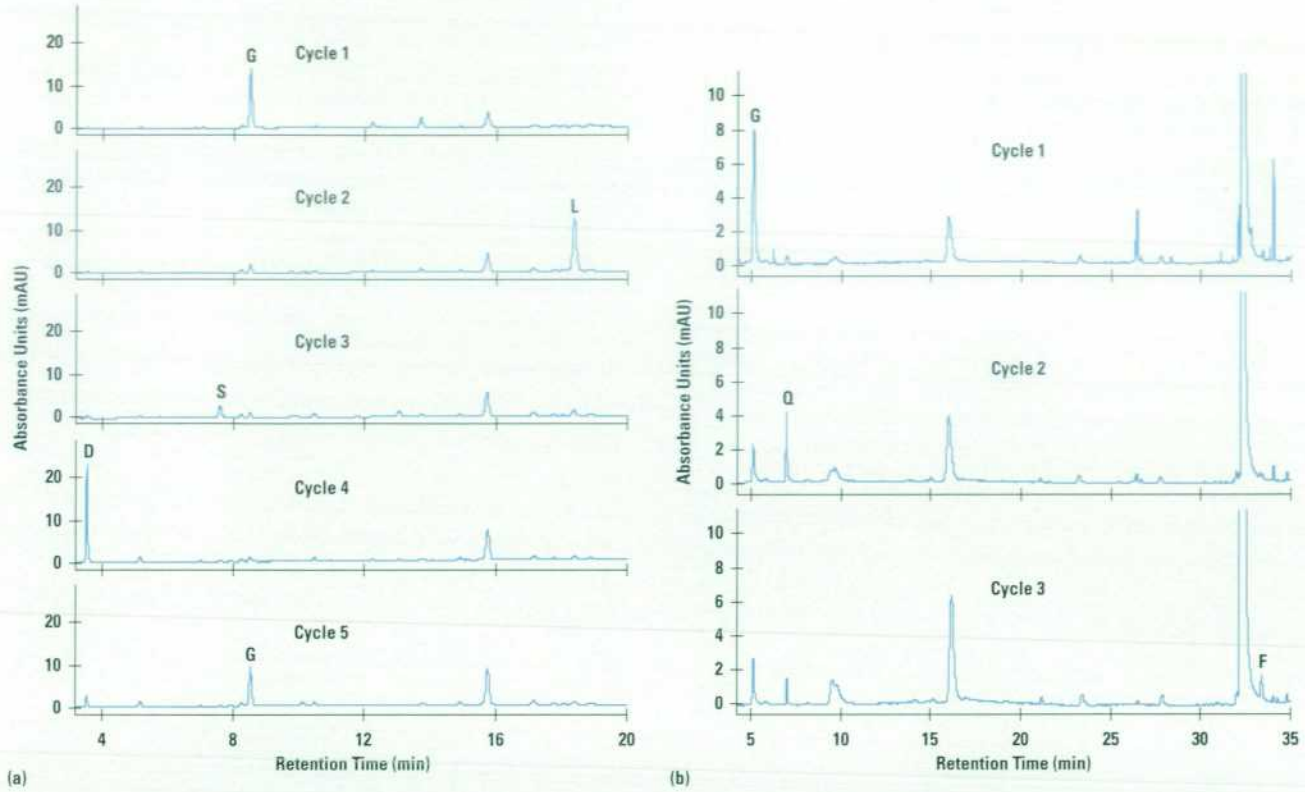


Fig. 17. (a) Cycles 1 to 5 of an N-terminal sequence analysis of 250 picomoles of horse apomyoglobin electroblotted from an SDS gel to Teflon tape. The HP G1005A N-terminal sequencer was used to sequence the sample applied to the Zitex membrane. 47 picomoles of Gly recovered at cycle 1. (b) Cycles 1 to 3 of the C-terminal sequence analysis of the N-terminal-sequenced myoglobin sample of Fig. 17a subjected to C-terminal sequencing using the HP G1009A C-terminal sequencer. The sample membrane was directly transferred from the HP G1005A N-terminal sequencer to the HP G1009A C-terminal sequencer. 79 picomoles of Gly recovered at cycle 1.

protein sequencer. The sample membrane is subsequently transferred to a C-terminal sequencer column and subjected to automated C-terminal sequence analysis using the HP G1009A C-terminal protein sequencer. In this fashion, a single sample is analyzed by both sequencing protocols, resulting in structural information that pertains precisely to a given population of proteins.[9]

Figs. 17a and 17b demonstrate the tandem sequencing protocol on an SDS gel sample of horse myoglobin. About 250 picomoles of myoglobin were applied to an SDS minigel across five lanes, subjected to electrophoresis, and electroblotted to a Teflon membrane. Five sulforhodamine-B-stained bands were excised and inserted into the N-terminal sequencer column. Five cycles of N-terminal sequencing identified the sequence Gly(1)-Leu(2)-Ser(3)-Asp(4)-Gly(5) with 47 picomoles of Gly recovered at cycle 1 as shown in Fig. 17a. The sequenced Teflon bands were transferred to a C-terminal sequencer column and the results of Fig. 17b show the expected C-terminal sequence Gly(1)-Gln(2)-Phe(3) with 79 picomoles of Gly recovered at cycle 1. The HP tandem sequencing protocol is currently being employed to ascertain the primary structure and sample homogeneity of pharmaceutical and biotechnology protein products in addition to protein systems under active study in academic research laboratories.

Summary

The HP G1009A C-terminal protein sequencer generates peptide and protein C-terminal amino acid sequence information on a wide variety of sample types and sample preparation techniques, including low-level sample amounts (< 100 picomoles), HPLC fractions, SDS gel electroblotted samples, and samples dissolved in various buffer solutions and formulations. The automated sequence analysis provides unambiguous and definitive amino acid residue assignments for the first cycle of sequence analysis and enables additional residue assignments for typically three to five cycles. The ability to sequence through any of the common amino acid residues provides researchers with a critical component for reliable sequence identification. The tandem process of N-terminal and C-terminal sequence analysis of single protein samples provides a significant solution for the evaluation of sample homogeneity and protein processing. The utility of these current applications is being applied to the investigation of many protein systems of importance in biochemistry and in the development and characterization of protein therapeutics.

Acknowledgments

The authors wish to thank James Kenny, Heinz Nika, and Jacqueline Tso of HP for their technical contributions, and our scientific collaborators including William Burkhart

and Mary Moyer of Glaxo Wellcome Research Institute, Research Triangle Park, North Carolina, Richard Simpson and colleagues of the Ludwig Institute for Cancer Research, Melbourne, and Ellie Ehrenfeld and Oliver Richards of the Department of Biological Chemistry, University of California, Irvine, California.

References

1. R. Harris, "Processing of C-terminal lysine and arginine residues of proteins isolated from mammalian cell culture," *Journal of Chromatography A*, Vol. 705, 1995, pp. 129-134.

2. P. Edman, "Method for determination of the amino acid sequence in peptides," *Acta Chemica Scandinavica*, Vol. 4, 1950, pp. 283-293.

3. P. Edman and G. Begg, "A protein sequenator," *European Journal of Biochemistry*, Vol. 1, 1967, pp. 80-91.

4. A.S. Inglis, "Chemical procedures for C-terminal sequencing of peptides and proteins," *Analytical Biochemistry*, Vol. 195, 1991, pp. 183-196.

5. J.M. Bailey, "Chemical methods of protein sequence analysis," *Journal of Chromatography A*, Vol. 705, 1995, pp. 47-65.

6. C.G. Miller and J.M. Bailey, "Automated C-terminal analysis for the determination of protein sequences," *Genetic Engineering News*, Vol. 14, September 1994.

7. G-F Tu, G.E. Reid, J-G Zhang, R.L. Moritz, and R.J. Simpson, "C-terminal extension of truncated recombinant proteins in Escherichia coli with a 10Sa RNA decapeptide," *Journal of Biological Chemistry*, Vol. 270, 1995, pp. 9322-9326.

8. W. Burkhart, M. Moyer, J.M. Bailey, and C.G. Miller, "C-terminal sequence analysis of proteins electroblotted to Teflon tape and membranes," Poster 529M, *Ninth Symposium of the Protein Society*, Boston, Massachusetts, 1995.

9. C.G. Miller and J.M. Bailey, "Expanding the scope of C-terminal sequencing," *American Biotechnology Laboratory*, October 1995.

Measuring Parasitic Capacitance and Inductance Using TDR

Time-domain reflectometry (TDR) is commonly used as a convenient method of determining the characteristic impedance of a transmission line or quantifying reflections caused by discontinuities along or at the termination of a transmission line. TDR can also be used to measure quantities such as the input capacitance of a voltage probe, the inductance of a jumper wire, the end-to-end capacitance of a resistor, or the effective loading of a PCI card. Element values can be calculated directly from the integral of the reflected or transmitted waveform.

by David J. Dascher

Why would anyone use TDR to measure an inductance or capacitance when there are many inductance-capacitance-resistance (LCR) meters available that have excellent resolution and are easy to use? First of all, TDR allows measurements to be made on devices or structures as they reside in the circuit. When measuring parasitic quantities, the physical surroundings of a device may have a dominant effect on the quantity that is being measured. If the measurement cannot be made on the device as it resides in the circuit, then the measurement may be invalid. Also, when measuring the effects of devices or structures in systems containing transmission lines, TDR allows the user to separate the characteristics of the transmission lines from the characteristics of the device or structure being measured without physically separating anything in the circuit. To illustrate a case where TDR can directly measure a quantity that is very difficult to measure with an LCR meter, consider the following example.

A printed circuit board has a long, narrow trace over a ground plane, which forms a microstrip transmission line. At some point, the trace goes from the top of the printed circuit board, through a via, to the bottom of the printed circuit board and continues on. The ground plane has a small opening where the via passes through it. Assuming that the via adds capacitance to ground, a model of this structure would be a discrete capacitance to ground between the top and bottom transmission lines. For now, assume that the characteristics of the transmission lines are known and all that needs to be measured is the value of capacitance to ground between the two transmission lines.

Using an LCR meter, the total capacitance between the trace-via-trace structure and ground can be measured but the capacitance of the via cannot be separated from the capacitance of the traces. To isolate the via from the traces, the traces are removed from the board. Now the capacitance between just the via and ground can be measured. Unfortunately, the measured value is not the correct value of capacitance for the model.

Using TDR instead of an LCR meter, a step-shaped wave is sent down the trace on the printed circuit board and the wave that gets reflected from the discontinuity caused by the via is observed. The amount of "excess" capacitance caused by the via can be calculated by integrating and scaling the reflected waveform. Using this method, the measured value of capacitance is the correct value of capacitance to be used in the model.

The discrepancy between the two measurements exists because the LCR meter was used to measure the total capacitance of the via while TDR was used to measure the excess capacitance of the via. If the series inductance of the via were zero, then its total capacitance would be the same as its excess capacitance. Since the series inductance of the via is not zero, a complete model of the via must include both its series inductance and its shunt capacitance. Assuming that the via is capacitive, the complete model can be simplified accurately by removing the series inductance and including only the excess capacitance rather than the total capacitance.

It should be no surprise that the value of excess capacitance measured using TDR is the correct value for the model. The reason to model the trace-via-trace structure in the first place is to predict what effect the via will have on signals propagating along the traces. TDR propagates a signal along the trace to make the measurement. In this sense, TDR provides a direct measurement of the unknown quantity.

To derive expressions that relate TDR waveforms to excess capacitance and inductance, an understanding of fundamental transmission line parameters is required. This article presents a cursory review of transmission lines and the use of TDR to characterize them, and then derives expressions for excess capacitance and inductance. If you are already familiar with transmission lines and TDR, you may wish to skip the review sections.

Fundamental Transmission Line Parameters

First, a few words about "ground." Twin-lead (or twisted-pair) wire forms a two-conductor transmission line structure that can be modeled as shown in Fig. 1. The model includes the

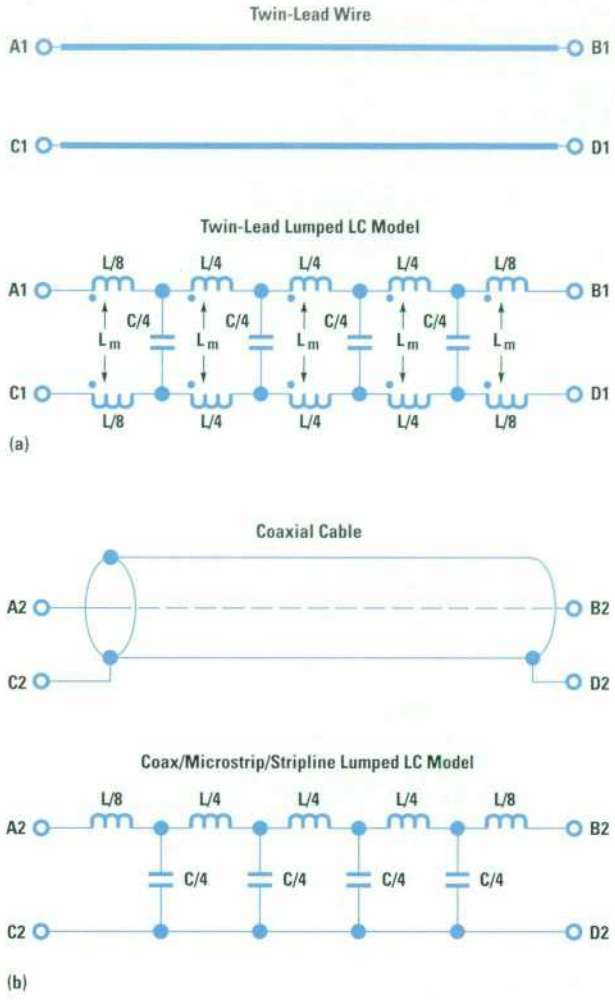


Fig. 1. Two-conductor transmission lines and lumped LC models.

self-inductance of each conductor, mutual inductance between the self-inductances, and capacitance between the two conductors. Skin effect and dielectric loss are assumed to be negligible in this model. Injecting a current transient i into one side of the transmission line, from node A1 to node C1, causes a voltage $v = iZ_0$ to appear between nodes A1 and C1 and also causes a voltage $v = Ldi/dt$ to appear across the series inductance of both the A-B and C-D conductors. Referring back to the physical structure, this means that there is a voltage difference between the left side and the right side of each conductor. Even if you name the C-D conductor "ground," it still develops a voltage between its left side and its right side, across its series inductance.

Microstrip traces and coaxial cables (coax) are two special cases of two-conductor transmission line structures. Injecting a current transient into one side of a microstrip or coax transmission line causes a voltage to appear across only one of the two conductors. In the case of an ideal microstrip, where one of the conductors is infinitely wide, the wide conductor can be thought of as a conductor with zero inductance. Hence the voltage generated across the infinitely wide conductor is zero. With coax, the inductance of the outer conductor is not zero. However, the voltage generated across the inductance of the outer conductor has two components. One component is a result of the current through the self-inductance of the outer conductor ($v_1 = L_{oc}di/dt$).

The other component is a result of the current through the center conductor and the mutual inductance between the center and outer conductors ($v_2 = L_m di/dt$). Current that enters the center conductor returns through the outer conductor, so the two currents are equal but in opposite directions. The unique property of coax is that the self-inductance of the outer conductor is exactly equal to the mutual inductance between the center and the outer conductors. Hence the two components that contribute to the voltage generated across the inductance of the outer conductor exactly cancel each other and the resulting voltage is zero. When current is injected into a coax transmission line, no voltage is generated along the outer conductor if the current in the center conductor is returned in the outer conductor.

The point here is that the generalized model for a two-conductor transmission line can be simplified for microstrip and coax constructions. The simplified model has zero inductance in series with one of the conductors since, in both cases, no voltage appears across the conductor. This conductor is commonly referred to as the ground plane in microstrip transmission lines and as the shield in coax transmission lines. The other conductor, the one that develops a voltage across it, is referred to as the transmission line, even though it is really only half of the structure.

There are two ways to model a lossless transmission line. One method defines the transmission line in terms of characteristic impedance (Z_0) and time delay (t_d) and the other

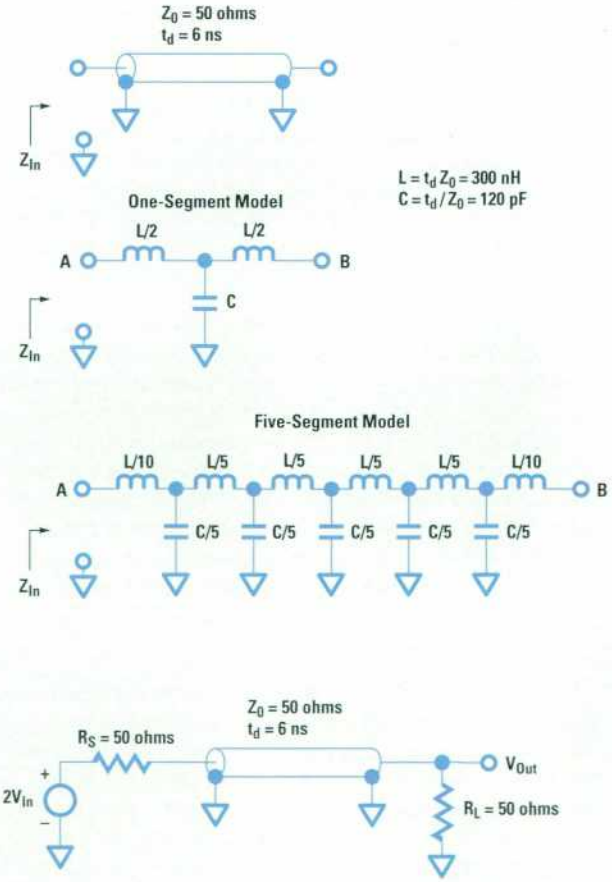


Fig. 2. Two LC models for a coaxial cable. The five-segment model is accurate over a wider range of frequencies than the one-segment model.

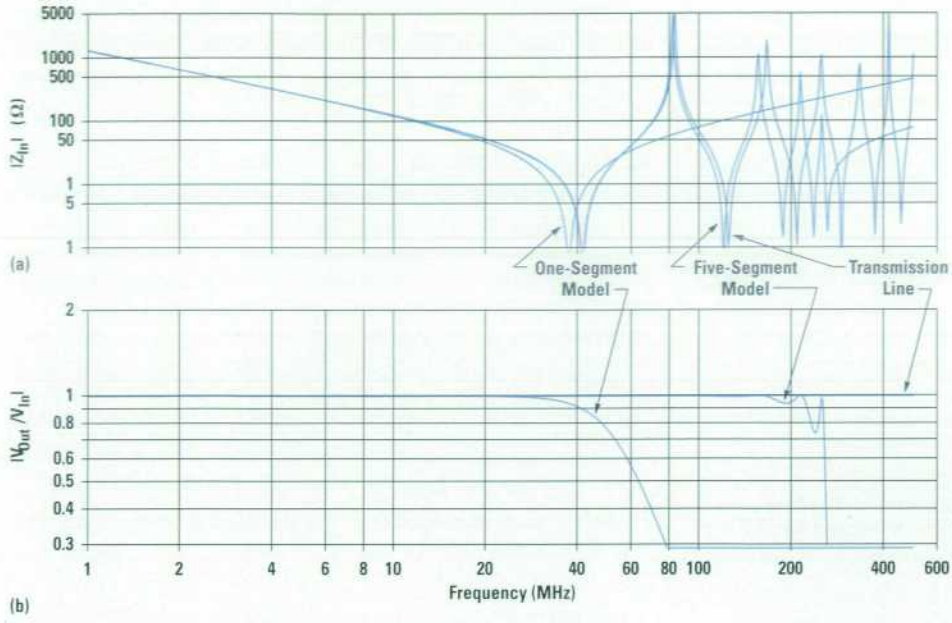


Fig. 3. (a) Magnitude of the input impedance Z_{In} for the transmission line and models of Fig. 2. (b) $|V_{Out}/V_{In}|$ for the transmission line and models of Fig. 2 (in the circuit at the bottom of Fig. 2).

method defines the transmission line in terms of total series inductance (L_{Total}) and total shunt capacitance (C_{Total}). There are times when it is beneficial to think of a transmission line in terms of Z_0 and t_d and likewise, there are times when thinking in terms of C_{Total} and L_{Total} is best.

If one end of a long transmission line is driven with an ideal current step, the voltage at that end will be an ideal voltage step whose height is proportional to the current and the characteristic impedance Z_0 of the transmission line: $V_{In} = I_{In}Z_0$. The waveform generated at that end will propagate along the transmission line and arrive at the opposite end some time later. The time it takes the wave to propagate from one end to the other is the time delay (t_d) of the transmission line.

The total capacitance and inductance of a transmission line can be measured with an LCR meter. To determine the total capacitance of a coaxial cable, measure the capacitance between the center conductor and the shield at one end of the cable while the other end is left open. The frequency of the test signal used by the LCR meter should be much less than $1/(4t_d)$ where t_d is the time delay of the cable. To measure the total inductance of the cable, measure the inductance from the center conductor to the shield at one end of the cable while the other end of the cable has the center conductor connected to the shield. Again, the test frequency must be much less than $1/(4t_d)$.

If the characteristic impedance and time delay of a transmission line are known, then the total shunt capacitance and total series inductance of the transmission line can be calculated as:

$$C_{Total} = \frac{t_d}{Z_0}$$

$$L_{Total} = t_d Z_0.$$

If the total shunt capacitance and total series inductance are known, then Z_0 and t_d can be calculated as:

$$Z_0 = \sqrt{\frac{L_{Total}}{C_{Total}}}$$

$$t_d = \sqrt{L_{Total} C_{Total}}.$$

As an example, many engineers use 50-ohm coaxial cables that are about four feet long and have BNC connectors on each end. The time delay of these cables is about six nanoseconds, the capacitance is 6 ns/50 ohms = 120 picofarads, and the inductance is 6 ns × 50 ohms = 300 nanohenrys.

Fig. 2 shows two LC models for a 50-ohm, 6-ns-long coaxial cable. The distributed inductance of the transmission line has been collected into two discrete inductors in the first model and six discrete inductors in the second model. The distributed capacitance has been collected into one discrete capacitor in the first model and five discrete capacitors in the second. Collecting the distributed capacitance and inductance into discrete, lumped elements reduces the range of frequencies over which the model is accurate. The more discrete segments there are, the wider the range of frequencies over which the model is accurate. Fig. 3 shows the magnitude of the impedance seen looking into one end of each model while the other end is left open. The five-segment model is accurate over a wider range of frequencies than the one-segment model.

Fig. 3 also shows the transmitted response through each of the models in a circuit that is both source and load terminated in 50 ohms. The discrete LC models are both low-pass filters. Again, the five-segment model is accurate over a wider range of frequencies than the one-segment model. In the time domain, a good rule of thumb is to break a transmission line into five segments per rise time. For example, to build a model that will accurately simulate the response of a step with a 1-ns rise time, the model should contain five segments per nanosecond of time delay. This would require a 30-segment model for the 4-foot, 6-ns coaxial cable.

The transmission line model used in SPICE and many other time-domain simulators is defined by characteristic impedance and time delay. It consists only of resistors, dependent sources, and time delay. It is an accurate model of a lossless transmission line over an infinite range of frequencies. There are three situations in which lumped LC models may be preferred over the Z_0-t_d model used by SPICE. When modeling very short sections of transmission lines, the maximum time increment is never greater than the time delay of the shortest transmission line. Short transmission lines can cause lengthy simulations. When modeling skin effect and dielectric loss, discrete LC models can be modified to include these effects. And finally, discrete LC models can be used to model transmission line systems that contain more than two conductors.

Characteristic impedance and time delay, or series inductance and shunt capacitance, completely specify the electrical properties of a lossless transmission line. Propagation velocity and length are also required to specify a physical cable or printed circuit board trace completely. If the time delay (also known as the electrical length) of a cable is 6 ns and the physical length of the cable is 4 feet, then the propagation velocity is 0.67 feet per nanosecond. The propagation delay is 1.5 ns per foot.

Reflection Coefficients for Impedance Changes

Fig. 4 shows a typical measurement setup using an oscilloscope with an internal TDR. The oscilloscope input is a 50-ohm transmission line terminated in 50 ohms. The TDR step generator can be modeled as a current source that connects to the 50-ohm transmission line between the front of the oscilloscope and the 50-ohm termination. When the current source transitions from low to high at time = 0, voltages are generated in the system as shown. Note that the upper traces in Fig. 4 are plots of voltage as a function of position along the transmission lines, not time. The waveform viewed by the oscilloscope is the waveform at the 50-ohm termination, which is shown at the bottom of Fig. 4. This is the TDR waveform.

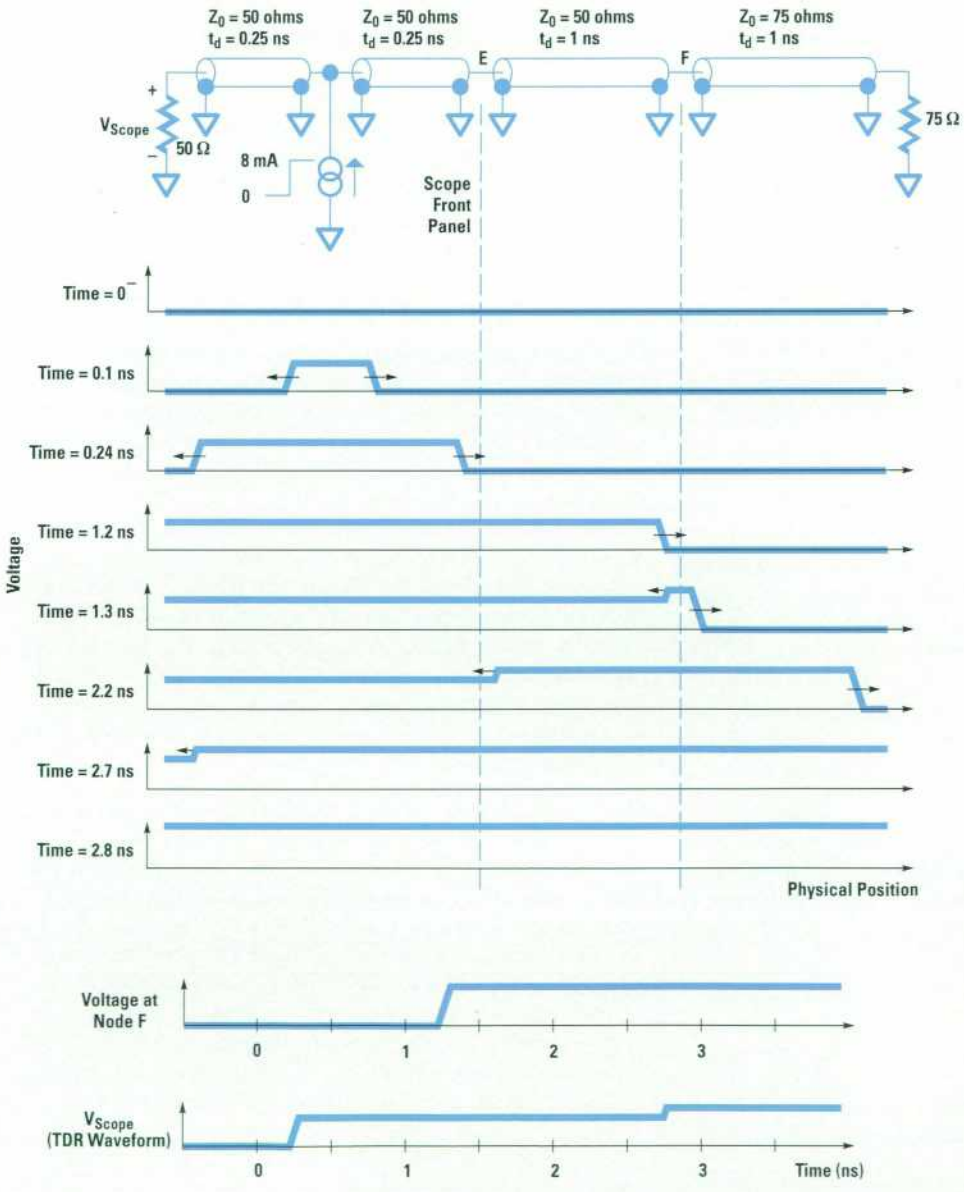


Fig. 4. Typical measurement setup and waveforms for an oscilloscope with an internal time-domain reflectometer (TDR).

There is only one discontinuity in the system: the transition between the 50-ohm cable and the 75-ohm cable at node F. When the wave propagating along the 50-ohm transmission line arrives at the 75-ohm transmission line, reflected and transmitted waves are generated. The wave that is incident on the transition is shown at time = 1.2 ns. Waves that are reflected and transmitted from the transition are shown at time = 1.3 ns. Defining a positive reflected wave to be propagating in the opposite direction from the incident wave, the reflected wave is given as:

$$\frac{\text{reflected}}{\text{incident}} = \frac{Z_B - Z_A}{Z_B + Z_A}.$$

where Z_A is the characteristic impedance through which the incident wave travels and Z_B is the characteristic impedance through which the transmitted wave travels. The transmitted wave is given as:

$$\frac{\text{transmitted}}{\text{incident}} = \frac{2Z_B}{Z_B + Z_A}.$$

If Z_A = 50 ohms, Z_B = 75 ohms, and the incident waveform is a 1V step, then the reflected waveform will be a step of height $1(75 - 50)/(75 + 50) = 0.2$ volt, and the transmitted waveform will be a step of $1(2 \times 75)/(75 + 50) = 1.2$ volts.

It is important to note that the waveform that appears at node F is the waveform that is transmitted into the 75-ohm cable. The waveform at node F can be derived by simplifying the circuit at the transition. Looking back into the 50-ohm cable, the circuit can be modeled as a voltage source in series with a resistor. The source is twice the voltage of the step propagating along the 50-ohm cable and the resistor is the characteristic impedance of the 50-ohm cable. Looking into the 75-ohm cable, the circuit can be modeled as a 75-ohm resistor. The simplified circuit is just a voltage divider in which the divided voltage is $2V_{Step}Z_2/(Z_2 + Z_1) = 1.2V_{Step}$. This is the transmitted waveform.

The TDR waveform, the waveform at the 50-ohm termination of the oscilloscope, is shown at the bottom of Fig. 4. The incident step is shown at time = 0.25 ns and the reflected step is shown at time = 2.75 ns. The characteristic impedance of the 75-ohm transmission line can be calculated from the TDR waveform by solving for Z_B in the above relationship, which gives:

$$Z_B = Z_A\left(\frac{\text{incident} + \text{reflected}}{\text{incident} - \text{reflected}}\right).$$

Both the incident and reflected waveforms are step functions that differ only in amplitude and direction of propagation. In the expression for Z_B, the step functions cancel and all that remains is the step heights. The TDR waveform in Fig. 4 shows the incident step height to be +1V and the reflected step height to be +0.2V. Knowing that the impedance looking into the oscilloscope is Z_A = 50 ohms, the characteristic impedance of the second cable, Z_B, can be calculated as $Z_B = 50(1V + 0.2V)/(1V - 0.2V) = 75$ ohms.

There is a "transition" between the front panel of the oscilloscope and the first cable. The characteristic impedance on either side of the transition is the same, so the reflected wave has zero amplitude. If there is no reflected wave, the transmitted wave is identical to the incident wave. Electrically, there is no discontinuity at this transition since it causes no reflection and the transmitted wave is identical to the incident wave.

The transition between transmission lines of different characteristic impedances is a discontinuity that produces reflected and transmitted waves that are identical in shape to the incident wave. The reflected wave differs from the incident wave in amplitude and direction of propagation. The transmitted wave differs only in amplitude. Discontinuities that are not simply impedance changes will produce reflected and transmitted waves that are also different in shape from the incident wave.

Discrete Shunt Capacitance

Fig. 5 shows two 50-ohm transmission lines with a shunt capacitor between them. The capacitance produces a discontinuity between the two transmission lines. A wave that is incident on the discontinuity produces reflected and transmitted waves. In this case, both the reflected and transmitted waves have a different shape than the incident wave.

Given an incident step, the voltage at node H can be ascertained by simplifying the circuit as shown in Fig. 5. The incident step, shown at time = 1.8 ns, has an amplitude of inc_ht and is propagating along a 50-ohm transmission line. Looking

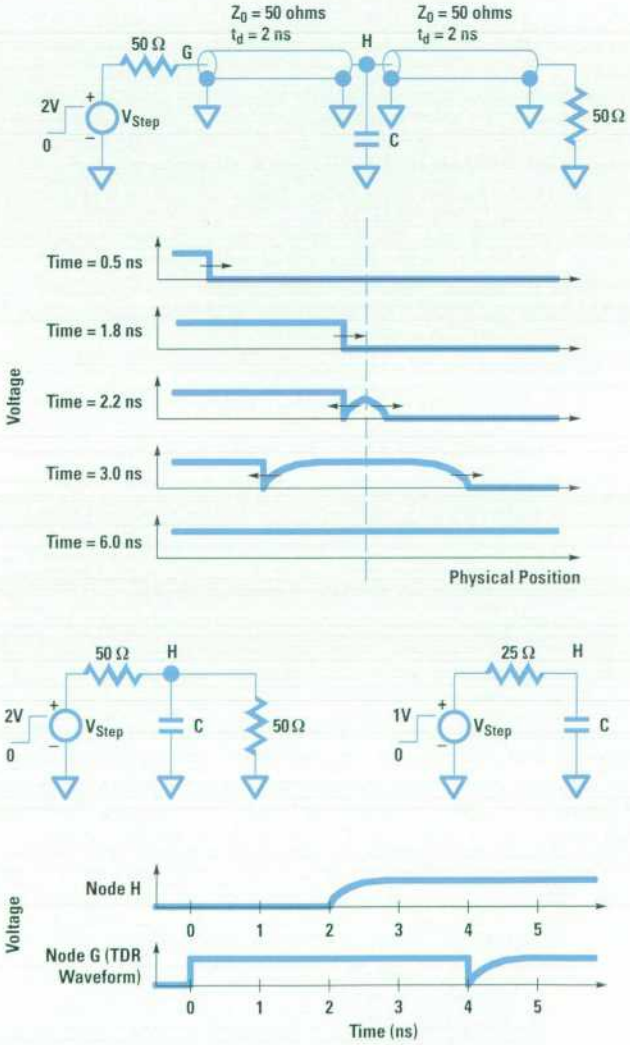


Fig. 5. Two 50-ohm transmission lines with a shunt capacitor between them. The capacitance C is to be measured.

to the left of the capacitor, the circuit can be simplified as a voltage source in series with a resistor. The voltage source is twice the incident voltage and the resistance is $Z_0 = 50$ ohms (not $R_S = 50$ ohms). Looking to the right of the capacitor, the circuit can be simplified as a resistor of value $Z_0 = 50$ ohms (not $R_L = 50$ ohms). The simplified circuit contains no transmission lines. The circuit can again be simplified by combining the two resistors and the voltage source into their Thevenin equivalent.

Note that if the left transmission line did not have a source resistance of 50 ohms or the right transmission line did not have a load resistance of 50 ohms, the simplified model would still be valid for 4 ns after the incident step arrives at node H. After 4 ns, reflections from the source and load terminations would have to be accounted for at node H.

Looking at the simplified circuit and redefining time = 0 to be the time when the incident step arrives at node H, the voltage at node H is zero for time < 0. For time > 0, the voltage at node H is:

$$v_H = \text{inc\_ht}\left(1 - e^{-\frac{t}{\tau}}\right),$$

where $\tau = RC = \frac{Z_0}{2}C$.

The voltage waveform that appears on node H is the transmitted waveform. The reflected wave can be determined knowing that the incident wave equals the transmitted wave plus the reflected wave (defining each wave as positive in its own direction of propagation). For time < 0, the reflected wave is zero. For time > 0, the reflected wave is:

$$\text{reflected} = -\text{inc\_ht} \cdot e^{-\frac{t}{\tau}},$$

where $\tau = RC = \frac{Z_0}{2}C$.

Normalizing the reflected waveform to the incident step height and integrating the normalized reflected waveform gives:

$$\text{reflected\_n} = \frac{\text{reflected}}{\text{inc\_ht}}$$

$$\int_{-\infty}^{+\infty} \text{reflected\_n} \cdot dt = \int_{0}^{+\infty} -e^{-\frac{t}{\tau}} dt = \tau e^{-\frac{t}{\tau}} \Big|_{0}^{+\infty} = -\tau.$$

Solving for the shunt capacitance between the two transmission lines gives:

$$C = \frac{2\tau}{Z_0} = -\frac{2}{Z_0} \int_{0}^{+\infty} \text{reflected\_n} \cdot dt.$$

Thus, the amount of capacitance between the two transmission lines can be determined by integrating and scaling the normalized reflected waveform.

Fig. 6 shows a measured TDR waveform from a 50-ohm microstrip trace with a discrete capacitor to ground at the middle of the trace. The TDR waveform has been normalized to have an incident step height of 1. The TDR waveform shows the incident step at 0.15 nanosecond and the reflected wave superimposed on the incident step at 2.3 nanoseconds. The reflected wave can be ascertained from the TDR waveform to be the TDR waveform minus one for all time after the incident step has settled. The bottom trace is the integral of the normalized reflected waveform scaled by $-2/Z_0 = -1/25$. Mathematically, the reflected waveform needs to be integrated until time = infinity, but practically, the reflected waveform only needs to be integrated until it has settled reasonably close to zero. In this case it takes about

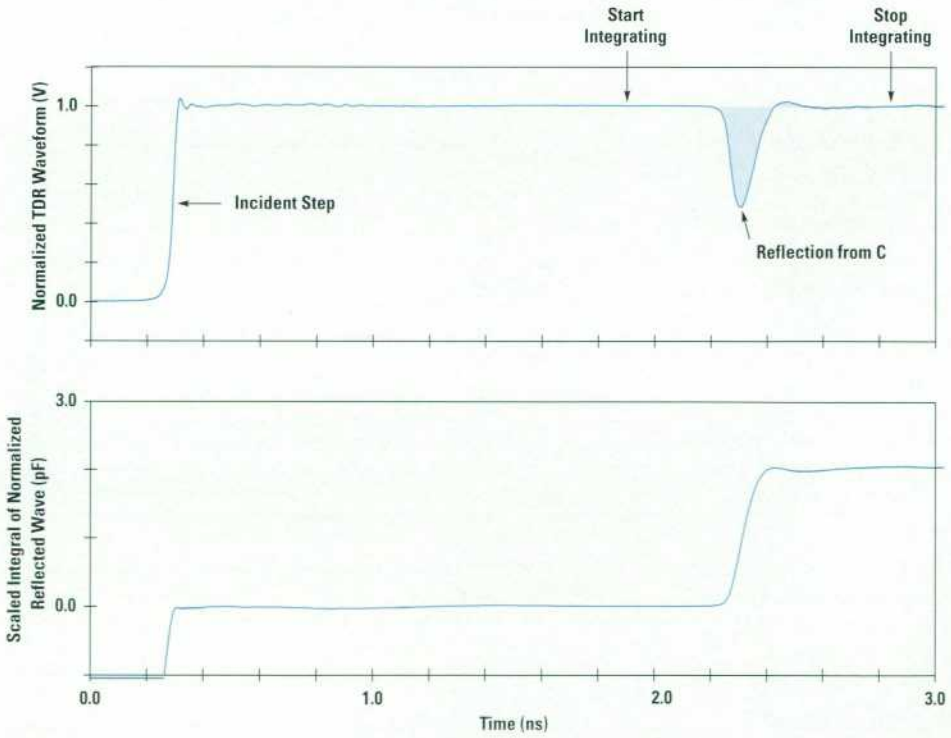


Fig. 6. (top) Measured TDR waveform from a 50-ohm microstrip trace with a discrete capacitor to ground at the middle of the trace. (bottom) The integral of the reflected waveform reveals that the capacitance is 2.0 pF.

500 picoseconds for the reflected wave to settle. The bottom trace shows that the discontinuity is a 2.0-pF capacitance.

Fig. 5 shows an ideal reflected wave to be an exponential with the same height as the incident step, but Fig. 6 shows a measured reflected wave to be similar to an exponential but with a rounded leading edge and a height much less than that of the the incident step. The measured reflected wave differs from the ideal reflected wave because of the nonzero rise time of the incident step and loss in the transmission lines. Neither of these nonideal conditions affects the integral of the reflected wave, as will be shown later.

Discrete Series Inductance

Fig. 7 shows two 50-ohm transmission lines with a series inductor between them. Given an incident step, the circuit can be simplified as shown to determine the voltage waveforms at nodes J and K. The voltages at nodes J and K are zero for time < 0. For time > 0, the voltages at nodes J and K are:

$$v_J = \text{inc\_ht}\left(1 + e^{-\frac{t}{\tau}}\right)$$

$$v_K = \text{inc\_ht}\left(1 - e^{-\frac{t}{\tau}}\right)$$

where $\tau = \frac{L}{R} = \frac{L}{2Z_0}$.

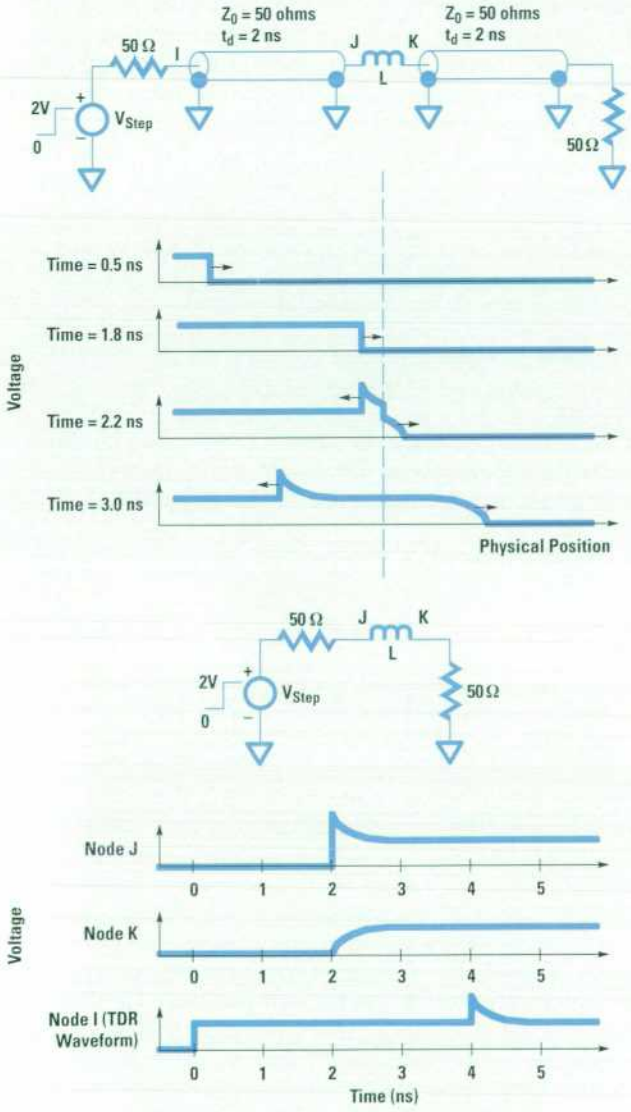


Fig. 7. Two 50-ohm transmission lines with a series inductor between them. The inductance L is to be measured.

In the previous case of a shunt capacitor, there was only one node between the two transmission lines at which the relationship incident = transmitted + reflected was applied. Now there are two nodes, each of which has unique incident, transmitted, and reflected waveforms. The voltage waveform at node J is the waveform that is transmitted to the inductor and the voltage waveform at node K is the waveform that is transmitted to the right transmission line. The waveform that is reflected at node J is the incident waveform minus the waveform transmitted to the inductor, that is, v_J.

For time < 0, the reflected wave is zero. For time > 0, the reflected wave is:

$$\text{reflected} = \text{inc\_ht} \cdot e^{-\frac{t}{\tau}}.$$

Normalizing the reflected waveform to the incident step height and integrating the normalized reflected waveform gives:

$$\text{reflected\_n} = \frac{\text{reflected}}{\text{inc\_ht}}$$

$$\int_{-\infty}^{+\infty} \text{reflected\_n} \cdot dt = \int_{0}^{+\infty} e^{-\frac{t}{\tau}} dt = -\tau e^{-\frac{t}{\tau}} \Big|_{0}^{+\infty} = \tau.$$

Solving for the series inductance between the two transmission lines gives:

$$L = 2Z_0\tau = 2Z_0 \int_{0}^{+\infty} \text{reflected\_n} \cdot dt.$$

Thus, the amount of inductance between the two transmission lines can be determined by integrating and scaling the normalized reflected waveform.

Excess Series Inductance

In the previous two cases of shunt capacitance and series inductance, the discontinuity was caused by discrete, lumped capacitance or inductance. Discontinuities can also be caused by distributed shunt capacitance and series inductance. Consider the printed circuit board traces shown in Fig. 8. Being over a ground plane, the traces form microstrip transmission lines. The circuit can be modeled as two long, 1-ns, 50-ohm transmission lines separated by a short, 100-ps, 80-ohm transmission line. Modeling the discontinuity as a short, high-impedance transmission line produces accurate simulations up to very high frequencies, or for incident waveforms with very fast transitions.

The discontinuity can also be modeled as a single discrete inductor. Modeling the discontinuity as an inductor produces less-accurate simulations at high frequencies, but when transition times are much larger than the time delay of the discontinuity, a simple inductor produces accurate simulation results. The simpler model provides quick insight into the behavior of the circuit. To say a discontinuity looks like 4.9 nH of series inductance provides most people with more insight than to say it looks like a 100-ps-long, 80-ohm transmission line. Also, simulating a single inductor is much faster than simulating a very short transmission line.

The total series inductance of the 100-ps-long, 80-ohm transmission line is:

$$L_{Total} = t_d Z_{High} = 100\ ps \times 80\ ohms = 8\ nH.$$

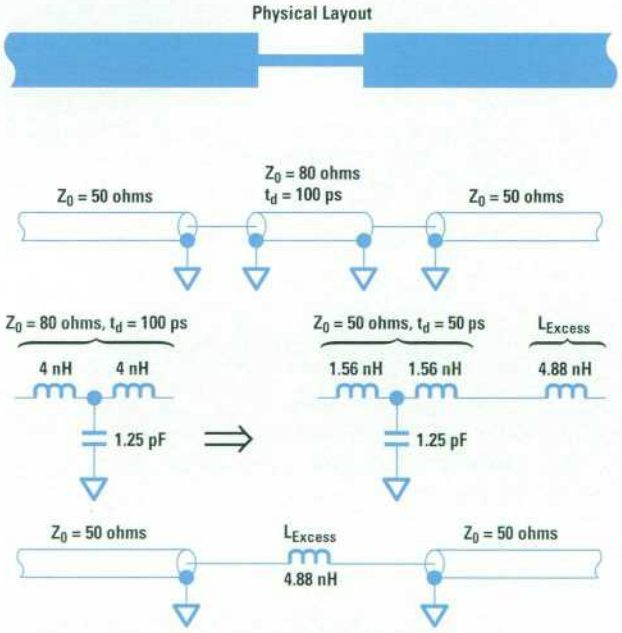


Fig. 8. Two 50-ohm microstrip transmission lines with a distributed discontinuity between them. The discontinuity can be modeled as a short 80-ohm transmission line or (less accurately) as a series inductor. The correct inductance to be used to model the narrow trace is L_{Excess}. (If the discontinuity were much wider than the 50-ohm traces, instead of narrower as shown here, it could be modeled as a shunt capacitor.)

To model the short trace as an 8-nH inductor would be incorrect because the trace also has shunt capacitance. The total shunt capacitance of the trace is:

$$C_{Total} = \frac{T_d}{Z_{High}} = \frac{100\ ps}{80\ ohms} = 1.25\ pF.$$

To simplify the model of the discontinuity, separate the total series inductance into two parts. One part is the amount of inductance that would combine with the total capacitance to make a section of 50-ohm transmission line, or "balance" the total capacitance. The remaining part is the amount of inductance that is in excess of the amount necessary to balance the total capacitance. Now, combine the total capacitance with the portion of the total inductance that balances that capacitance and make a short section of 50-ohm transmission line. Put this section of transmission line in series with the remaining, or excess, inductance. The short 50-ohm transmission line does nothing except increase the time delay between the source and load. The excess inductance (not the total inductance) causes an incident wave to generate a nonzero reflected wave and a nonidentical transmitted wave. This is the correct amount of inductance to use to model the short, narrow trace.

In this case, the portion of the total inductance that balances the total capacitance is:

$$L_{Balance} = C_{Total} Z_0^2 = 1.25\ pF \times (50\ ohms)^2 = 3.125\ nH$$

and the remaining, excess inductance is:

$$L_{Excess} = L_{Total} - L_{Balance}$$
$$= 8\ nH - 3.125\ nH = 4.875\ nH.$$

Expressing the excess inductance in terms of the time delay and impedance of the narrow trace (t_d, Z_{High}) and the impedance of the surrounding traces ($Z_0 = Z_{Ref}$) gives:

$$L_{Excess} = L_{Total} - L_{Balance}$$
$$= t_d Z_{High} - C_{Total} Z_{Ref}^2 = t_d Z_{High} - \frac{t_d}{Z_{High}} Z_{Ref}^2$$

$$L_{Excess} = t_d Z_{High} \left(1 - \left(\frac{Z_{Ref}}{Z_{High}} \right)^2 \right).$$

Fig. 9a shows TDR waveforms from a SPICE simulation that compare the reflection generated from an ideal 4.88-nH inductor to the reflection generated from an 80-ohm, 100-ps transmission line, both in a 50-ohm system. Fig. 9b shows TDR waveforms of the same circuit as Fig. 9a, but the rise time of the incident step in Fig. 9b is longer than the rise time used in Fig. 9a. Notice that when the rise time of the incident step is much larger than the time delay of the discontinuity, the simplified model agrees with the transmission line model. As the rise time of the incident step changes, the peak value of the reflected waveform also changes but the integral of the reflected waveform does not change. Using the integral rather than the peak value to quantify the discontinuities makes the measurement of excess inductance rise time independent.

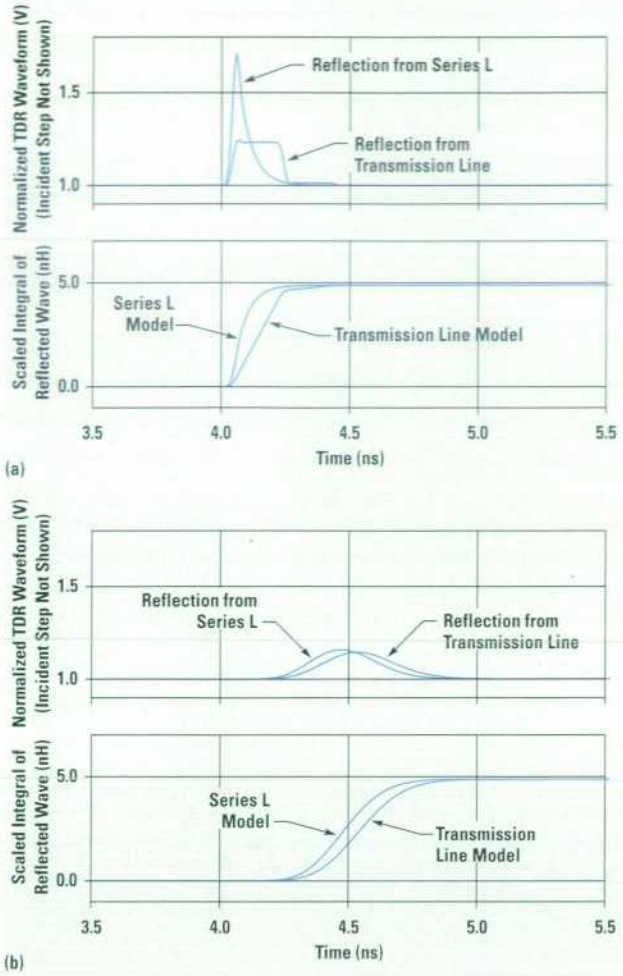


Fig. 9. (a) TDR waveforms from a SPICE simulation showing reflections from an ideal 4.88-nH inductor and an 80-ohm, 100-ps transmission line, both in a 50-ohm system. The final value of the integral of the reflected wave is the same for both. (b) Same as (a) for an incident step with a longer rise time. The agreement between the reflected waves for the single-inductor model and the transmission line is better. The final value of the integral of the reflected wave is unchanged.

Excess Shunt Capacitance

In the example shown in Fig. 8, the discontinuity is a section of trace that is narrower than the surrounding traces. This discontinuity can be modeled as an inductor. If a discontinuity is a section of trace that is wider than the surrounding traces, the discontinuity can be modeled as a capacitor. As in the previous case, the correct value of capacitance is not the total capacitance of the wide section of trace but rather the difference between the total capacitance and the portion of that capacitance that combines with the total inductance to make a section of transmission line of the same impedance as the surrounding, reference impedance. In terms of the time delay and impedance of the wide trace (t_d, Z_{Low}) and the impedance of the surrounding traces ($Z_0 = Z_{Ref}$), the excess capacitance is:

$$C_{Excess} = \frac{t_d}{Z_{Ref}}\left(1 - \left(\frac{Z_{Low}}{Z_{Ref}}\right)^2\right).$$

The length of the transmission line to the right of the discontinuity has no effect on the reflected or transmitted waveforms since it is terminated in its characteristic impedance. It has been shown as a long transmission line so the transmitted wave can be viewed easily. If the right transmission line is removed then the discontinuity is at the termination of the left transmission line. Quantifying parasitic inductance and capacitance at the termination of a transmission line is no different from quantifying parasitic inductance and capacitance along a transmission line.

Non-50-Ohm Z_{Ref}

Often, the impedance of the system to be measured is not the same as the impedance of the TDR/oscilloscope system. There are two ways to deal with non-50-ohm systems. First, the TDR can be connected directly to the system to be measured. Second, an impedance matching network can be inserted between the TDR and the system to be measured. In either case, the above relationships can be applied if corrections are made to the TDR waveform.

Connecting the TDR directly to a non-50-ohm system, as shown in Fig. 10, creates a system with two discontinuities, one being the transition between the 50-ohm TDR and the non-50-ohm system and the other being the discontinuity to be measured. The incident step will change amplitude at the non-50-ohm interface before arriving at the discontinuity to be measured. The reflected wave from the discontinuity creates another reflection when it arrives at the non-50-ohm-to-50-ohm interface. This secondary reflection propagates back into the non-50-ohm system and generates another reflection when it arrives at the discontinuity being measured. If twice the time delay of the transmission line between the non-50-ohm interface and the discontinuity being

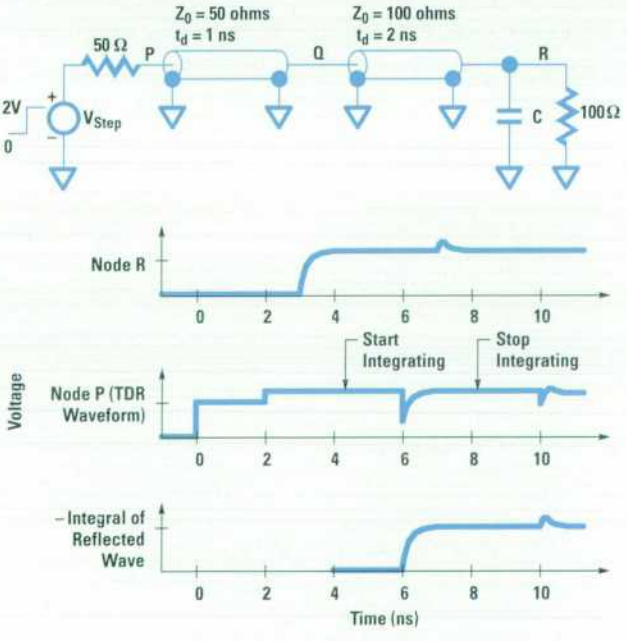


Fig. 10. A TDR connected to a non-50-ohm transmission line with a capacitance C to be measured. There are multiple reflections. The discontinuity can be accurately quantified as long as the reflection to be measured can be isolated from other reflections on the TDR waveform.

measured is larger than the settling time of the first reflected wave, then the integral of the normalized reflected wave can be evaluated before any secondary reflections arrive. Otherwise, all of the secondary reflections must settle before the integral can be evaluated. If the discontinuity is purely capacitive or inductive, then the rereflections will not change the final value to which the integral converges.

Two corrections need to be made to the TDR waveform. First, the height of the step that is incident on the discontinuity, inc_ht, is not the same as the height of the step that was generated by the TDR, tdr_ht. Second, the reflected waveform that is viewed by the oscilloscope, reflected_scope, has a different magnitude from the waveform that was reflected from the discontinuity, reflected. The relationships between the characteristic impedance of the system being measured, the characteristic impedance of the TDR, and the above waveforms are:

$$\text{inc\_ht} = \text{tdr\_ht}\frac{2Z_{Ref}}{Z_{Ref} + Z_{TDR}}$$

$$\text{reflected\_scope} = \text{reflected}\frac{2Z_{TDR}}{Z_{TDR} + Z_{Ref}}.$$

The characteristic impedance of the system being measured can be calculated from the height of the step reflected from the transition between the 50-ohm TDR system and the non-50-ohm (Z_{Ref}) system, ref1_ht.

$$Z_{Ref} = Z_{TDR}\left(\frac{\text{incident} + \text{reflected}}{\text{incident} - \text{reflected}}\right)$$

$$= 50\ \text{ohms}\left(\frac{\text{tdr\_ht} + \text{ref1\_ht}}{\text{tdr\_ht} - \text{ref1\_ht}}\right).$$

To calculate series inductance or shunt capacitance, the integral of the waveform reflected from the discontinuity, normalized to the height of the step incident on the discontinuity, needs to be evaluated. In terms of the measured TDR waveform, the normalized reflected waveform is:

$$\text{reflected\_n} = \frac{\text{reflected}}{\text{inc\_ht}} = \frac{\text{reflected\_scope}\left(\frac{Z_{Ref} + Z_{TDR}}{2Z_{TDR}}\right)}{\text{tdr\_ht}\left(\frac{2Z_{Ref}}{Z_{Ref} + Z_{TDR}}\right)}$$

$$= \frac{\text{reflected\_scope}}{\text{tdr\_ht}}\frac{(Z_{Ref} + Z_{TDR})^2}{4Z_{Ref}Z_{TDR}}.$$

As an example, the shunt capacitance would be:

$$C = -\frac{2}{Z_{Ref}}\int_0^{+\infty} \text{reflected\_n} \cdot dt$$

$$= -\frac{(Z_{Ref} + Z_{TDR})^2}{2Z_{TDR}Z_{Ref}^2}\int_0^{+\infty}\frac{\text{reflected\_scope}}{\text{tdr\_ht}}dt.$$

Reflections between the discontinuity and the non-50-ohm transition can be removed if an impedance matching network is inserted between the TDR and the non-50-ohm system. Two resistors are the minimum requirement to match an impedance transition in both directions. Attenuation across the matching network needs to be accounted for before integrating the reflected waveform. When a matching network is used, neither the impedance of the system being tested nor the attenuation of the matching network can be ascertained from the TDR waveform. If tran_12 is the gain of the matching network going from the TDR to the system and tran_21 is the gain going from the system to the TDR, then the normalized reflected waveform is:

$$\text{reflected\_n} = \frac{\text{reflected}}{\text{inc\_ht}} = \frac{\left(\frac{\text{reflected\_scope}}{\text{tran\_21}}\right)}{\text{tdr\_ht} \cdot \text{tran\_12}}$$

$$= \frac{\text{reflected\_scope}}{\text{tdr\_ht}}\frac{1}{\text{tran\_12} \cdot \text{tran\_21}}.$$

Series Capacitance

So far, all of the discontinuities that have been measured have been mere perturbations in the characteristic impedance of transmission line systems. The discontinuities have been quantified by analyzing the reflected waveforms using TDR. Now consider a situation where a capacitance to be measured is in series with two transmission lines, as shown in Fig. 11. The series capacitance can be quantified using the transmitted waveform rather than the reflected waveform,

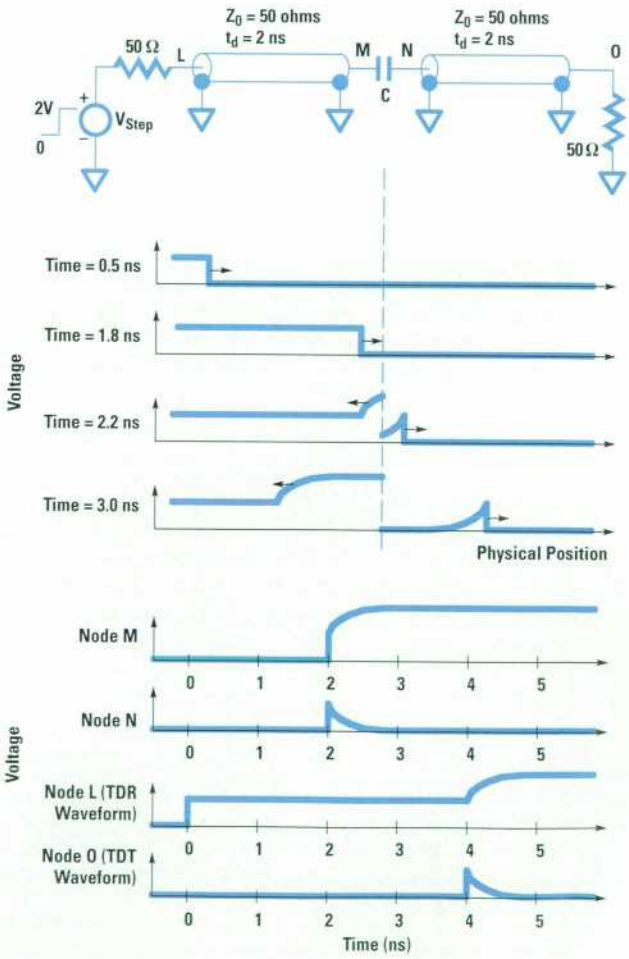


Fig. 11. A series capacitance between two transmission lines can be measured using the transmitted waveform (time-domain transmission or TDT) rather than TDR.

or using *time-domain transmission* (TDT). Given an incident step that arrives at the series capacitance at time = 0, the transmitted waveform is zero for time < 0. For time > 0, the transmitted waveform is:

$$\text{transmitted} = \text{inc\_ht} \cdot e^{-\frac{t}{\tau}},$$

where $\tau = RC = 2Z_0C$.

Note that τ is now proportional to $2Z_0$ instead of $Z_0/2$. Normalizing the transmitted waveform to the incident step height and integrating the normalized, transmitted waveform gives:

$$\text{transmitted\_n} = \frac{\text{transmitted}}{\text{inc\_ht}}$$

$$\int_{-\infty}^{+\infty} \text{transmitted\_n} \cdot dt = \int_{0}^{+\infty} e^{-\frac{t}{\tau}} \cdot dt = -\tau e^{-\frac{t}{\tau}} \Big|_{0}^{+\infty} = \tau.$$

Solving for the series capacitance gives:

$$C = \frac{\tau}{2Z_0} = \frac{1}{2Z_0} \int_{0}^{+\infty} \text{transmitted\_n} \cdot dt.$$

Shunt Inductance

The last topology to be analyzed will be shunt inductance to ground. Given an incident step that arrives at the shunt inductance at time = 0, the transmitted waveform is zero for time < 0. For time > 0, the transmitted waveform is:

$$\text{transmitted} = \text{inc\_ht} \cdot e^{-\frac{t}{\tau}},$$

where $\tau = \frac{L}{R} = \frac{L}{(Z_0/2)} = \frac{2L}{Z_0}$.

This is the same transmitted waveform as in the case of series capacitance. τ is $2L/Z_0$, so the shunt inductance is:

$$L = \frac{Z_0}{2}\tau = \frac{Z_0}{2} \int_{0}^{+\infty} \text{transmitted\_n} \cdot dt.$$

More Complex Discontinuities

Consider a circuit in which a transmission line is brought to the input of an integrated circuit (IC) and terminated (Fig. 12). Looking from the termination into the IC, there might be inductance of the lead in series with a damping resistance in series with the input capacitance of the IC. A first-order approximation of the load that the IC puts on the termination of the transmission line is simply a capacitor. The reflected wave can be integrated to calculate the capacitance, but will the series inductance and resistance change the measured value? Looking back to Fig. 5, the current into the shunt capacitor is:

$$I = \frac{2\text{inc\_ht}}{Z_0} e^{-\frac{t}{\tau}} = -\text{reflected}\frac{2}{Z_0}.$$

The integral of the reflected wave is directly proportional to the integral of the current into the capacitor. This means

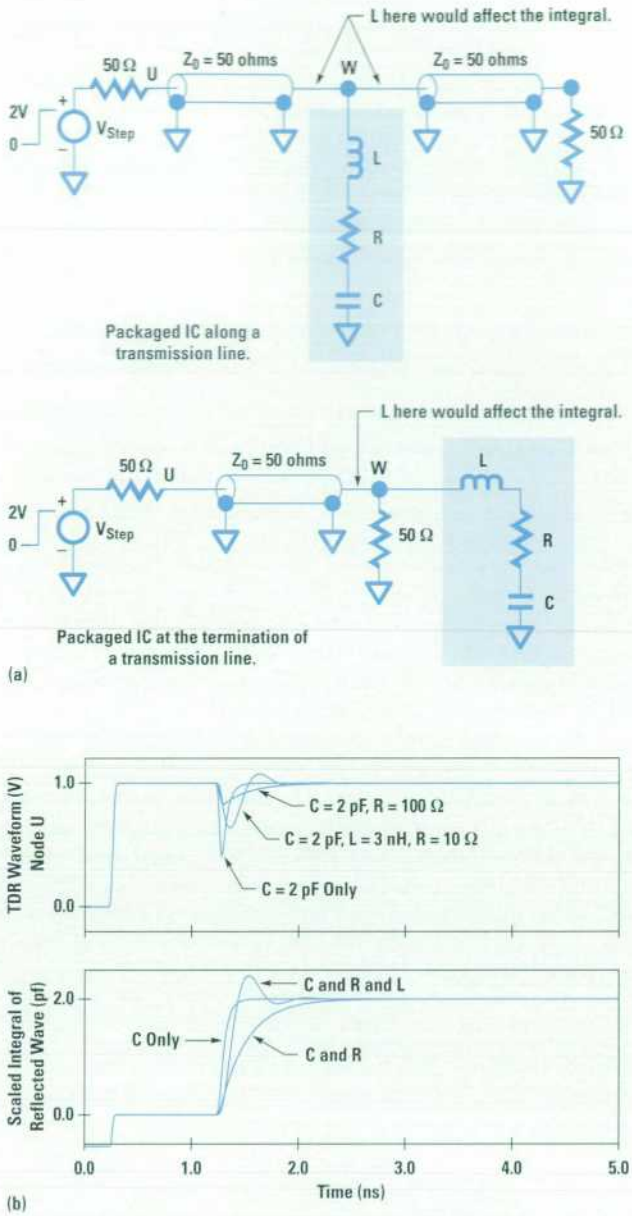


Fig. 12. L and R in series with a shunt capacitance do not affect the measurement of C.

that the value of capacitance calculated from integrating the reflected wave is proportional to the total charge on the capacitor. Resistance or inductance in series with the shunt capacitor does not change the final value of charge on the capacitor. Hence, these elements do not change the measured value of capacitance. Series resistance decreases the height and increases the settling time of the reflected wave. Series inductance superimposes ringing onto the reflected waveform. Neither changes the final value of the integral of the reflected wave. Note that inductance in series with the transmission line, as opposed to inductance in series with the capacitor, will change the measured value of capacitance. Similarly, when measuring series inductance, resistance or capacitance in parallel with the series inductance does not change the measured value of inductance.

When building a model for the input of the packaged IC, integrating the reflected wave provides the correct value of capacitance for a single capacitor model and also for higher-order models. If, for example, a higher-order model contains a section of transmission line terminated with some capacitive load, then the value of capacitance measured is the total capacitance of the transmission line plus the load. To ascertain values for L and R in Fig. 12, a more rigorous analysis of the TDR waveform must be performed.

Now consider a situation in which a voltage probe is used to measure signals on printed circuit board traces. What effect does the probe have on signals that are propagating along the traces? To a first-order approximation, the load of a high-impedance probe will be capacitive and the value of capacitance can be measured by integrating the reflected wave. However, some passive probes that are optimized for bandwidth can also have a significant resistive load (these probes usually have a much lower capacitive load than high-impedance probes). Looking at the reflected wave in Fig. 13, the resistive load of the probe introduces a level shift at 4 ns. The level shift adds a ramp to the integral so that the integral never converges to a final value. To measure the value of capacitance, the integral can be evaluated at two different times after the reflected wave has settled to its final value, say at 6 and 7 ns. Since the reflected wave has settled, the difference between the two values is caused only by the level shift. The component of the integral resulting from the level shift can be removed from the total integral by subtracting twice the difference between the values at 6 and 7 ns from the value at 6 ns. The remaining component of the integral is a result of the shunt capacitance.

Multiple Discontinuities

Sometimes, the transmission lines between the TDR/oscilloscope and the discontinuity to be measured are less than ideal. For example, the connection between a 50-ohm coaxial cable and a 50-ohm printed circuit board trace can look

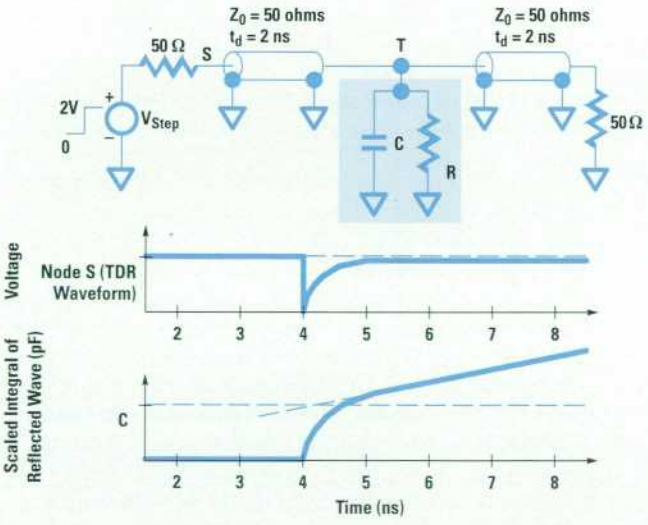


Fig. 13. A voltage probe used to measure signals on printed circuit board traces may act like a shunt RC load, adding a ramp to the integral of the reflected wave.

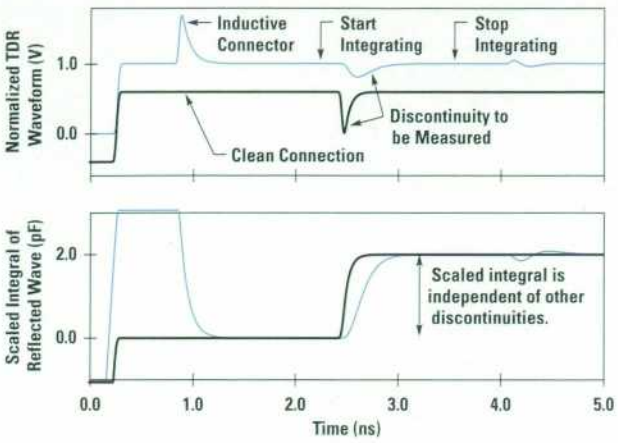


Fig. 14. Intermediate discontinuities do not affect the measurement as long as the discontinuity to be measured can be isolated on the TDR waveform.

quite inductive if a short jumper wire is used. Also, the system being measured may have more than one discontinuity.

Intermediate discontinuities between the TDR and the discontinuity to be measured act as low-pass filters that increase the rise times of both the incident step and the reflected wave. Intermediate discontinuities also cause secondary reflections in the TDR waveform, as shown in Fig. 14. If the discontinuity to be measured can be separated from surrounding discontinuities on the TDR waveform, then the discontinuity can be accurately quantified by integrating the reflected wave. Increasing the rise time of the system decreases any separation between discontinuities but, as was shown earlier, does not change the integral of the reflected wave.

Minimizing Effects of Discontinuities (Balancing)

There are many instances in which discontinuities in transmission line systems are unavoidable. The effects of an unavoidable discontinuity can be minimized by introducing discontinuities of opposite polarity as close as possible to the unavoidable discontinuity. For example, if the narrow part of the trace shown in Fig. 8 is required, then the traces on either side of the narrow trace can be made wider to compensate for the excess inductance of the trace. The amount of excess capacitance required to compensate for the 4.88 nH of excess inductance is $Z_0^2 L = (50 \text{ ohms})^2 \times 4.88 \text{ nH} = 1.95 \text{ pF}$. If the widest trace that can be added has a characteristic impedance of 30 ohms, then 152 ps of this trace will add 1.95 pF of excess capacitance. Propagation delays on typical printed circuit board traces are around 150 ps/inch, so adding 0.5 inch of 30-ohm transmission line on each side of the narrow trace will balance the excess inductance of the trace. Balancing the discontinuity reduces reflections and makes the transmitted wave more closely resemble the incident wave. A discontinuity that is balanced will produce a reflected wave whose integral is zero. In this case, the integral of the reflected wave can be used to determine how well a discontinuity is balanced.

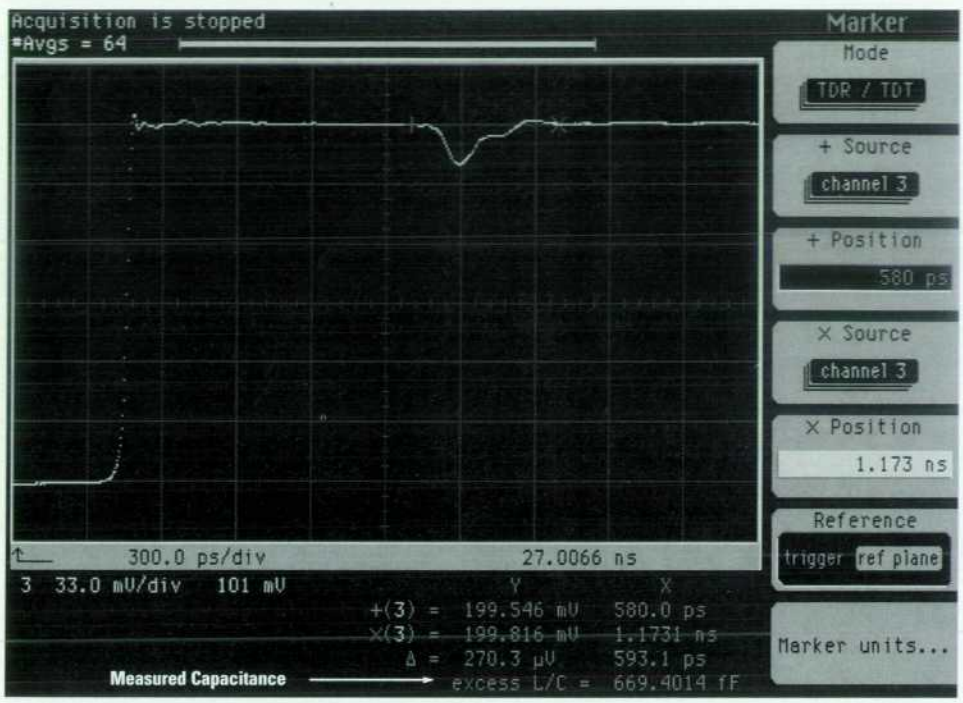


Fig. 15. Measuring the input capacitance of an HP 54701A active probe by placing cursors around the reflection caused by the probe.

Instrumentation

Hewlett-Packard recently introduced the HP 54750A oscilloscope and the HP 54754A TDR plug-in. All of the measurements discussed above can be made by positioning two cursors on the TDR or TDT waveform. For TDR, the position of the leftmost cursor defines when to start integrating the reflected wave and also determines the reference impedance for the calculation of excess L or C. The position of the rightmost cursor determines when to stop integrating the reflected wave. Likewise, when using TDT, the cursors determine when to start and stop integrating the transmitted wave. The oscilloscope calculates the integral of the normalized reflected or transmitted wave and displays the appropriate value of capacitance or inductance, as shown in Figs. 15 and 16. Parasitic inductances and capacitances can be accurately measured in the time it takes to position two cursors.

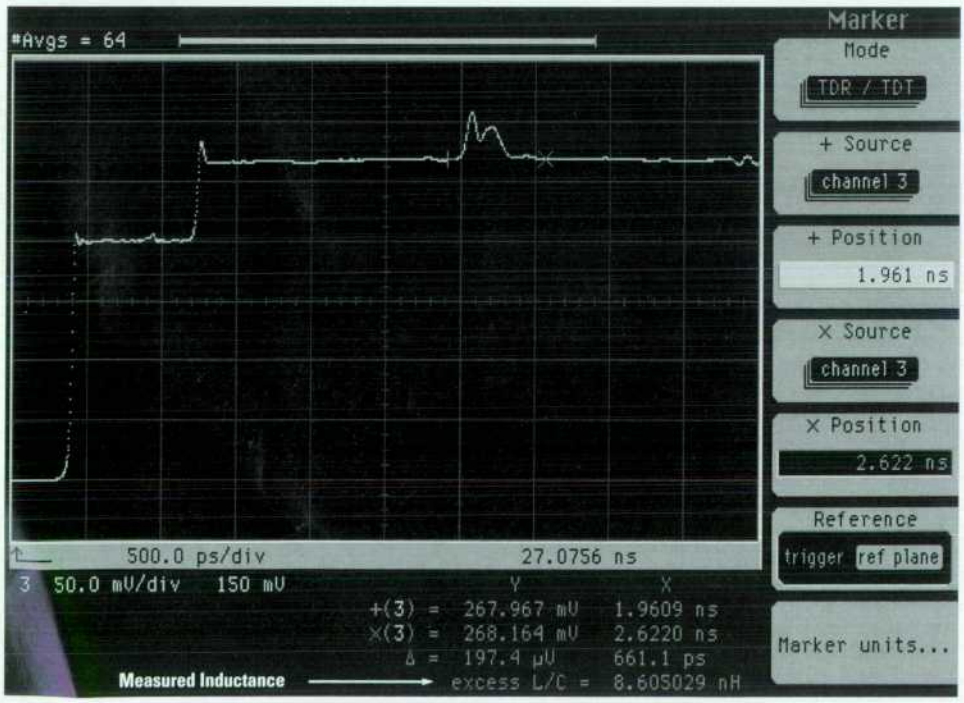


Fig. 16. Measuring the excess inductance of a connector in a 100-ohm system by placing cursors around the reflection caused by the inductance.

Summary

TDR can be used to measure impedances and discontinuities in transmission line systems very accurately. Discontinuities that are either primarily capacitive or primarily inductive can be accurately modeled by a single element whose value can be determined by integrating the reflected or transmitted waves. In 50-ohm systems, the integral of the normalized reflected wave is simply the time constant of the network.

Discontinuities can also be quantified in terms of the peak value of the reflections they generate. The problem with using the peak value of the reflection is that the value depends on the rise time of the incident step and the loss of the transmission line system between the TDR and the discontinuity. Using the integral of the reflected wave to quantify a discontinuity makes the measurement independent of the rise time of the incident step and also independent of discontinuities between the TDR and the discontinuity being measured, as long as the discontinuity being measured can be isolated from surrounding discontinuities on the TDR waveform.

Acknowledgment

The author would like to acknowledge the contributions of John Kerley who provided insight and direction in the modeling of transmission line systems.

Authors

April 1996

6 Common Desktop Environment

Brian E. Cripe

Brian Cripe is a software engineer at the Technical Computing Center, in HP's Corvallis lab. He joined HP in 1982 after graduating from Rice University with a BS degree in electrical engineering and a BA degree in computer science. Initially at HP he worked as a firmware engineer on the ThinkJet and DeskJet printers. He is named as an inventor in two patents related to printer firmware, including test scale mode and delta row data compression, and he authored an article about the DeskJet. He was responsible for representing HP's technical interest in the multicompany design process for CDE. He is currently responsible for investigating new middleware opportunities in 3D graphics. Brian was born in Annapolis, Brazil. He is married, recently celebrated the birth of his second daughter, and is considered to be the world champion swing pusher at his local park. He is also interested in all forms of cycling, especially mountain bike racing, and enjoys telemark skiing.

Jon A. Brewster

Jon Brewster is an architect for enterprise visualization at HP's Corvallis lab. He was a section manager for the early CDE work and HP VUE. Before that, he was the project manager for the early X Window System products. He also acted as project lead and architect for HP's first window systems. He graduated in 1980 from Oregon State University with a BS degree in electrical and computer engineering and has worked at HP since that time except for a two-year stint in Hawaii where he worked on Mauna Kea telescopes. Jon is married and has five children. He loves astronomy and motorcycle riding.

Dana E. Laursen

Dana Laursen is a technical contributor at the Technical Computing Center at HP's Corvallis lab. He is currently an architect for desktop environments including HP CDE 1.0, digital media integration, and the developer's toolkit. He recently worked on the architecture for the sample implementation for CDE 1.0 and was HP's representative on the CDE change review and architecture teams. In addition, he was a major contributor to the X/Open standard and was the principal architect for default configuration. He came to HP in 1988. Initially he worked on user-interface toolkit projects including Motif 1.0 and projects investigating user-interface builders and HyperCard for the UNIX operating system. He is professionally interested in desktop environments, toolkits, builders, and the World-Wide Web. He has coauthored several papers about the Common Desktop Environment, the X Window System, and the use of computers in education. Before coming to HP, he worked as an architect for window system software at Tektronix. He was also an engineer for artificial intelligence languages and graphics and did artificial intelligence research at the University of Oregon. He taught computer science at Clackamas Community College, was a graduate teaching fellow in computer science and art education at the University of Oregon, and taught at Illinois Valley High School. He received a BA degree in art from Reed College in 1974. He did coursework at Lewis and Clark College for an Oregon teaching certificate. He graduated with an MS degree in art education in 1979 and an MS degree in computer science in 1984, both from the University of Oregon. Born in Minneapolis, Minnesota, Dana is married and has a daughter. His hobbies include hiking, gardening, and reading.

16 Applications in CDE

Anna Ellendman

Born in Philadelphia, Pennsylvania, Anna Ellendman received a BS degree in chemistry from Pennsylvania State University in 1970 and an MS degree in science education from Oregon State University in 1975. She was a chemistry instructor at Oregon State University and also at Linn Benton Community College. Joining HP's Corvallis Division in 1980, she wrote manuals for HP Series 80 computer products and for several financial and scientific calculators. After joining the Workstation Systems Division, she worked on OSF/Motif documentation and the system administration documentation for HP VUE. She was HP's project leader for the CDE documentation. Currently she is responsible for information access projects. Anna is a music enthusiast and her hobbies include playing guitar, piano, and harp. She also enjoys producing folk music concerts and is president of the Corvallis Folklore Society and a member of the Da Vinci Days entertainment committee.

William R. Yoder

A software project manager in the Corvallis software lab, Bill Yoder joined HP in 1980. He is currently responsible for enterprise connectivity solutions for workstation customers. He was responsible for the X11R6 X server, the X11R6 font server, the federal computer operation, and technical security. Previously he was responsible for client/server implementation, application integration, and desktop printing for the CDE project. When he first came to HP, he worked as a senior technical writer creating owner's manuals, programmer's manuals, and tutorials for such products as the HP 75C and HP 85B calculators, Word/80, CP/M, Series 80 p-system, and the HP Integral portable computer. He then became a lab engineer and worked on the X10 Window System, HP Vectra Pascal, HP VUE 2.0, and HP VUE 3.0. He was the project lead for the Starbase/X11 Merge server and HP MPower. He coauthored a paper on Starbase/X11 Merge and authored one on HP MPower. His professional interests include graphical user interfaces, security, distributed computing, and collaboration. He is a member of ACM, IEEE, USENIX, and CPSR (Computer Professionals for Social Responsibility). Bill was born in Bloomington, Illinois. He received a BA degree in history and literature in 1972 from Harvard University and an MA degree in English in 1975 from the University of California at Santa Barbara. He taught high school and college English classes in California and Oregon for six years. He also earned a BS degree in computer science in 1985 from Oregon State University and an MS degree in computer science in 1988 from Stanford University. He is married and has one child in high school. In his free time he enjoys mountain biking, rock guitar, fiction, and refereeing high school soccer.

25 Action and Data Typing Services

Arthur F. Barstow

Art Barstow joined HP's User Interface Division in 1988. He worked on HP VUE's infrastructure, test execution systems, and X server development. He is professionally interested in distributed computing systems and was responsible for the development and testing of the CDE remote execution system. He left HP in May 1995 to work for the X Consortium on the next generation of CDE. Before joining HP, he earned a BS degree in forest management from the University of Idaho in 1980. He then worked for the U.S. Forest Service as a forester and also worked with their computer mapping systems. He went on to earn a BS degree in computer science from Oregon State University in 1985. He was also an instructor there. Art was born in Baltimore, Maryland. He enjoys hiking, cycling, and birding and spending time with his two sons.

30 HP VUE Customizations

Molly Joy

Born in Kerala, India, Molly Joy received a BA degree in economics from the Stella Maris College in Madras, India, in 1979. She earned a BS degree in computer science from Oregon State University in 1985 and joined HP's Workstation Technology Division in 1988. She is currently a software development engineer at the Workstation Technology Center and is working on HP's CDE offering. She is responsible for several components, as well as the migration suite. For CDE 1.0, she developed an application that would allow HP VUE customers to move their VUE customizations to CDE. She designed and developed the framework, the user interface, and several of the tools for this project. Previously, she designed and developed various components for OSF/Motif toolkits 1.0, 1.1, and 1.2. She coauthored a paper on improving Motif test technology. Molly is married, has two children, and has worked part-time at HP since 1990 to spend time with her children. Some activities she enjoys with her family are biking and traveling.

39 CDE Help System

Lori A. Cook

Lori Cook is a software design engineer at the Technical Computing Center at HP's Corvallis lab. She is currently responsible for the next generation of CDE, including isolation and creation of APIs for sections of the CDE 1.0 help system. She joined HP's Personal Computer Division in June 1980, after completing her BS degree in computer science from Oregon State University where she majored in systems design. At HP, she initially worked on the mass storage, printer/plotter, and service ROMS for the HP 87 computer. She then worked on the electronic disk ROM for the HP 85 and 87 computers and on the mass storage driver for the HP 110 laptop computer. She also worked on the Pascal compiler for the HP Integral computer and the HP-IB commands library for the HP Vectra. For the HP-UX operating system, she worked on the first X11 Windows clients (HP-UX 6.5), the help server (HP VUE 2.0), and the help library (HP VUE 3.0). She coauthored an article on the HP VUE 3.0 help system. She worked on the help library for the CDE 1.0 help system and was responsible for the interface layer between the application and the help information, including reading help files, laying out text and graphics, and rendering displayed information. Lori was born in Tacoma, Washington. She is married and her hobbies include knitting, skiing, racquetball, and firearms competition.

Stephen P. Hiebert

A software design engineer at the Technical Computing Center at HP's Corvallis lab, Steve Hiebert is currently working on a DocBook-to-SDL (Semantic Delivery Language) translator for the next release of the Common Desktop Environment. For the CDE 1.0 project, he worked on the design of SDL and the HelpTag-to-SDL translator. Previously he worked on SGML (Standard Generalized Markup Language) for the HP VUE 3.0 help system and wrote the HelpTag-to-CCDF (Cache Creek delivery format) translator. Before that, he provided engineering support for the X toolkit, worked on the embedded C interpreter in the UIMX Interface Architect, and developed the MotifGen application, which generates standard output from the Interface Architect. He also worked on the X10 and X11 servers including the Starbase/X11 Merge server, developed the software engineering ROM for the HP Integral portable computer (IPC), and ported compilers and systems utilities to the IPC. He has coauthored articles on Starbase/ X11 Merge and the HP VUE 3.0 help system. He also authored an article on the design and implementation of a ROM-based set of UNIX utilities for the IPC. His professional interests include computer language translators, compilers, and interpreters, especially the use of these tools for SGML processing. He is a member of IEEE, ACM, and USENIX, and is a sponsoring member of the Davenport Group (the standards body for the DocBook SGML document type definition (DTD) for computer software documentation). Before joining HP's Portable Computer Division in 1981, he worked at Electro-Scientific Industries, Inc. designing and developing a commerical Pascal compiler. He was also a systems programming and engineering manager at TimeShare Corporation. Steve was born in Portland, Oregon and attended Portland State University, majoring in mathematics with a computer emphasis. He spent four years in the U.S. Air Force as an instructor of electronics communications and cryptographic equipment systems repair and achieved the rank of staff sergeant. He enjoys commuting to work on his bicycle.

Michael R. Wilson

Mike Wilson is a member of the technical staff at the Technical Computing Center at HP's Corvallis lab. He is currently the project lead responsible for maintaining and enhancing the lab's testing framework. He recently worked on the CDE help system and was responsible for product packaging, custom widget development, and test planning and implementation. Except for test planning, he did the same things for HP VUE. He has authored or coauthored three papers on help systems. He earned a BS degree in computer science from California State University at Chico and joined HP's Electronic Design Division in 1987. Mike is married, has two children, and his primary hobby is cycling.

51 Multicompany Development

Robert M. Miller

An R&D section manager in HP's Technical Computing Center at Corvallis, Bob Miller is currently responsible for technical enterprise connectivity, including CDE desktop technology, Web tools technology, collaboration, and security. He joined HP's Handheld Calculator Division in 1981. In R&D, he worked on software modules for the HP 41C series, including the plotter module about which he authored a paper. He also worked on software products for the HP 75 calculator, including the FORTH language and text formatter. He was also a team member for the HP 28C advanced scientific calculator and is named as a coinventor in a patent related to stacking, evaluating, and executing arbitrary objects. He also published a paper on the HP 28C. He was a product marketing engineer for high-end calculator products, after which he transferred to the Corvallis Workstation Technology Center where he was a project manager responsible for Motif 1.0, HP VUE 2.0, HP VUE 3.0, CDE joint development, and CDE 1.0. He received a BA degree in English literature in 1973 from LaSalle College and a BS degree in computer science in 1980 from California Polytechnic State University at San Louis Obispo. He also earned an MS degree in computer science in 1989 from the National Technology University. Born in Philadelphia, Pennsylvania, Bob is married and has two children. He enjoys reading and outdoor activities such as hiking, camping, and golf.

55 CDE Test Suite

Kristann L. Orton

A software development engineer at HP's Technical Computing Center at Corvallis, Kristann Orton is currently investigating 3D graphics middleware toolkits for use by ISVs (independent software vendors) and customers in the MCAD and ECAD markets. She joined HP in 1992, after completing a BS degree in computer science at the University of California, Santa Barbara. When she first came to HP, she worked on the Motif development team, then moved to a team formed to automate the validation of HP's release software. Afterwards, she became one of HP's representatives on the joint CDE test team. She worked on the joint development and implementation of the testing framework, oversaw test engineers, and ensured timely deliverables. Before coming to HP, Kristann worked as a securities trader in an investment counseling firm in Santa Barbara.

Paul R. Ritter

Paul Ritter earned a BS degree in computer science in 1970 from the University of California at Berkeley, where he also worked at the University's space science lab developing application software for digital analysis of remote sensing imagery. He started work at HP as a contractor in 1989. He was hired by the Workstation Technology Division as a regular employee in 1993 and earned an MS degree in computer science from Oregon State University the same year. He is currently a software development engineer at HP responsible for X server support. Previously, he worked on testing of Motif 1.0 and HP VUE 2.0, and recently helped develop the test suite for CDE 1.0. He has authored three papers, two on remote sensing algorithms and techniques and one on testing of Motif 1.0. He is a member of ACM and IEEE. Born in Washington state, Paul is married and has one child. His hobbies include skiing, softball, and motorcycles, and he is an avid reader of fiction and nonfiction books.

63 Synlib

Sankar L. Chakrabarti

Born in Azimganj, W. Bengal, India, Sankar Chakrabarti received a PhD degree in chemistry in 1974 from the Tata Institute of Fundamental Research in Bombay, India. In 1985 he received an MS degree in computer science from Oregon State University. He joined HP's Portable Computer Division in 1981 and is now a member of the technical staff at the Workstation Technology Center. He has worked on all aspects of the X Window System, has developed tools for automating the testing of this system, and is the developer of Synlib, the core tool used for Common Desktop Environment testing. He currently is responsible for new technology and processes to improve software testing and engineering. He has published several papers on software testing, GUI test tools, and program specification. He is the winner of the 1992 HP Computer Systems Organization quality and productivity award and is an invited speaker at conferences and universities on the topic of software testing and specification. Sankar is married and has two children. His wife is an ink chemist at HP's Inkjet Supplies Business Unit.

67 GSM Power Module

Melanie M. Daniels

A hardware design engineer at HP's Communication Components Division since 1991, Melanie Daniels is currently working on the design of base station amplifiers. She was a design engineer and team leader for development of the GSM power module described in this issue. She has authored a paper on a 1-watt low-cost power module, and her work on an ultralinear feed-forward base station amplifier has resulted in a pending patent. Before joining HP, she was a design engineer for modular components at Avantek. She was awarded a BS degree in microwaves and RF in 1986 from the University of California at Davis and an MSEE degree in digital communications in 1995 from California State University at Sacramento. Melanie is married. In her free time, she is a member of ToastMasters and enjoys traveling, camping, and reading.

74 Protein Sequence Analysis

Chad G. Miller

A product marketing manager at HP's California Analytical Division, Chad Miller is currently responsible for protein sequencing products. He has also served as product manager for the HP G1009A C-terminal sequencer and the HP G1004B protein chemistry station, and as an applications manager and applications chemist. He is professionally interested in protein chemistry, bioscience technologies, and bio-organic chemistry. He is a member of the U.S. Protein Society and of the American Association for the Advancement of Science. He has published numerous articles in the past five years on protein sequencing and analysis and on C-terminal analysis. He received a BS degree in chemistry in 1976 from the University of California at Los Angeles and an AM degree in chemistry in 1980 from Harvard University. Before coming to HP, he worked at the Beckman Research Institute of the City of Hope Medical Center as a senior research associate in the Division of Immunology. He joined HP's Scientific Instruments Division in 1990. Chad is married and has two children. His outside interests include creative writing, U.S. stamp collecting, backyard astronomy, and gourmet coffees.

Jerome M. Bailey

Jerome Bailey received a BS degree in biochemistry in 1982 from the University of Rochester and a PhD in chemistry in 1987 from the University of Delaware. Before joining HP, he was a tenure-track assistant research scientist at the Beckman Research Institute of the City of Hope Medical Center for seven years. His work focused on protein sequencing chemistry and hardware. He left the Institute and joined HP's Protein Chemistry Systems Division in 1994. As a research scientist, he is currently responsible for C-terminal sequencing of proteins and R&D. He is professionally interested in enzymology and protein characterization and is named as an inventor in twelve patents involving these areas of interest, especially C-terminal and N-terminal sequencing. He has also authored and coauthored numerous articles on these topics. He is a member of the American Chemical Society, the Protein Society, and the American Association for the Advancement of Science. Jerome was born in Southington, Connecticut. He is married and has one child.

84 Time-Domain Reflectometry

David J. Dascher

Dave Dascher graduated from Colorado State University with a BSEE degree. He joined HP's Colorado Springs Division in 1984 as a hardware design engineer. He is currently developing a high-density analog probe system. Previously he consulted on the HP 54754 TDR feature set and worked on the HP 54720 oscilloscope analog-to-digital converter (ADC) system and the HP 54111 oscilloscope ADC. His work has resulted in four pending patents related to probing.

HEWLETT-PACKARD
JOURNAL

April 1996 Volume 47 • Number 2

Technical Information from the Laboratories of
Hewlett-Packard Company

Hewlett-Packard Company, Hewlett-Packard Journal
3000 Hanover Street, Palo Alto, CA 94304-1185 U.S.A.

HEWLETT® PACKARD

5964-6218E