Tool Kit
User's Guide

Order No. AA-N617E-TK

November 1985

New Tool Kit users should read this manual first. It describes how to write applications for the Professional computer using the Tool Kit.

REQUIRED SOFTWARE: Host Tool Kit V3.0
or PRO/Tool Kit V3.0

OPERATING SYSTEM: P/OS V3.0

digital™

DIGITAL EQUIPMENT CORPORATION
Maynard, Massachusetts 01754-2571

First Printing, December 1982
Revised, May 1983
Revised, September 1983
Revised, April 1984
Revised, November 1985

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The specifications and drawings, herein, are the property of Digital Equipment Corporation and shall not be reproduced or copied or used in whole or in part as the basis for the manufacture or sale of items without written permission.

The following are trademarks of Digital Equipment Corporation:

CTI BUS	MASSBUS	Rainbow
DEC	PDP	RSTS
DECmate	P/OS	RSX
DECsystem-10	PRO/BASIC	Tool Kit
DECSYSTEM-20	PRO/Communications	UNIBUS
DECUS	Professional	VAX
DECwriter	PRO/FMS	VMS
DIBOL	PRO/RMS	VT
digital™	PROSE	Work Processor
	PROSE PLUS	

CONTENTS

FIGURES

TABLES

PREFACE

Manual Objectives

The *Tool Kit User's Guide* is your primary source of general information about the Tool Kit. After reading this manual, you will be able to begin writing applications for the Professional computer using Tool Kit software.

Appendix A contains information about other manuals in the document set.

Intended Audience

We assume that you are an experienced programmer, although you may not necessarily be familiar with the Professional computer or other DIGITAL products. If you are not familiar with the Tool Kit, you should read this manual first.

Also, you should be familiar with the end user documentation for the Professional and P/OS. If you are using the Host Tool Kit, you should be familiar with the host system environment.

Structure of This Document

The *Tool Kit User's Guide* contains the following chapters and appendices:

- Chapter 1, *Getting Started*, explains what the Tool Kit is all about. It provides an overview of the Tool Kit configurations, the target system, the development cycle, and describes how to invoke the Tool Kit.

- Chapter 2, *Designing the Application*, describes some of the factors to be considered during the design phase.

- Chapter 3, *Implementing the Application*, describes the source files that you must produce and the process that you follow to create the files. P/OS file specifications and logical names are described. An overview of the steps involved in creating a menu-driven user interface is provided.

- Chapter 4, *Building the Application*, provides a brief explanation of how to use the Professional Application Builder, the utility that creates task images.

- Chapter 5, *Testing the Application*, provides an explanation of the steps that you take to test your application on the Professional.

- Chapter 6, *Tuning the Application*, explains some factors that affect an application's performance.

- Appendix A, *Documentation Directory*, provides an abstract of each of the manuals in the Tool Kit documentation set.

- The *Glossary* describes terms used in this manual.

Associated Documents

This book makes frequent references to other books in the Tool Kit documentation set. Appendix A provides an abstract of each Tool Kit book.

You should be familiar with other documentation:

- **Professional 300 Series End User Documentation**

  Read this documentation to understand the user's view of the system on which your applications will run. We assume that you are familiar with the primary end-user documents.

- **Host System Documentation**

  Complete documentation for RSX-11M/M-PLUS and VAX/VMS host systems can be obtained under separate license. We assume that you are familiar with the manuals for your host system.

- **High-Level Language Documentation**

  Five optional high-level languages are supported: Tool Kit BASIC-PLUS-2, Tool Kit COBOL-81, Tool Kit DIBOL, Tool Kit FORTRAN-77, and Tool Kit PASCAL. See your Tool Kit Software Product Description (SPD) for information on optional software products and documentation.

Conventions Used in This Document

Convention/Term	Meaning
[optional]	In a command line, square brackets indicate that the enclosed item is optional. In a file specification, square brackets are part of the required syntax.
UPPERCASE	Uppercase words and letters indicate that you should type the word or letter exactly as shown.
lowercase	Lowercase words and letters indicate that you should substitute a word or value of your own. Usually the lowercase word identifies the type of substitution required.
...	A horizontal ellipsis indicates that you can repeat the preceding item one or more times. For example: parameter [,parameter...]
. . .	A vertical ellipsis means that not all of the statements are shown.
red	Interactive input appears in red.
Tool Kit	This general term refers to the software you use to develop applications to run on a Professional computer.
Host Tool Kit	The Host Tool Kit is Tool Kit software that runs on a host computer, rather than on the Professional itself.
PRO/Tool Kit	The PRO/Tool Kit is the Tool Kit software that runs on the Professional computer.

CHAPTER 1

GETTING STARTED

The Tool Kit is a collection of software that you use to develop applications for the Professional computer. Applications developed using the Tool Kit can run on P/OS, the Professional Operating System.

P/OS, as well as many of the Tool Kit components, is derived from software that runs on other members of the PDP-11 minicomputer family, including much larger systems like the PDP-11/44. For example, P/OS is based on the RSX-11M-PLUS operating system. Most of the Tool Kit software consists of layered RSX-11M-PLUS components that run on P/OS.

If you have had experience with PDP-11 minicomputers and RSX-11M-PLUS, you will find the Tool Kit very similar to tools that you have already used. However, if your background has been primarily with personal computers, you may find that the Tool Kit and P/OS are more powerful and complex than what you have been using.

This manual is your primary source of general information about the Tool Kit. First read this manual to learn about the Tool Kit, then refer to Appendix A for information about other manuals in the document set.

1.1 TWO TOOL KITS

There are two available Tool Kits:

- **PRO/Tool Kit**

  The PRO/Tool Kit runs as an installed application on the Professional computer and uses the DIGITAL Command Language (DCL) as its interface. It allows you to use the Professional as a development system and as a target system at the same time.

All the tools you need for program development are included in the PRO/Tool Kit.

To use the PRO/Tool Kit, start up your system and select the PRO/Tool Kit option from the appropriate menu. The dollar sign ($) prompt indicates that DCL is ready to accept commands. For complete information on DCL, refer to the *PRO/Tool Kit Command Language and Utilities Manual*.

- **Host Tool Kit**

  The Host Tool Kit runs on a VAX/VMS system as a layered product under VAX-11 RSX; it also runs on a PDP-11 system under RSX-11M/M-PLUS. You can take advantage of the performance, mass storage, and communications provided by a larger host system while using the Professional as a terminal and a target system.

  For host system work, you can use whatever terminal you prefer. We recommend that you use a VT200-type terminal or your Professional system running Terminal Emulator software. With such a terminal, you can work interactively with 8-bit characters. If you use a VT100-type terminal, you are limited in your ability to create and view the full 8-bit character set.

  Both VAX/VMS and RSX-11M/M-PLUS require that you enter some terminal commands to enable transmission or reception of 8-bit characters. See the *PRO/Communications Manual* for details on terminal commands.

Figure 1-1 illustrates the two Tool Kits.

1.2 TARGET SYSTEM CONFIGURATIONS

The Professional currently has three models: 325, 350 and 380. Each model provides a different level of flexibilty.

The three models, in combination with different varieties of P/OS, provide the following target Professional configurations:

- **Workstation**

  Any Professional can be a workstation running in a P/OS Server environment. Workstations do not require any storage media. (However, a floppy disk drive is recommended.) See Section 1.2.2 for information on P/OS Server.

- **Stand-alone with Hard Disk or Server**

  The 350 and 380 models can run P/OS stand-alone or P/OS Server, since these models have the required hard disk. Additionally, for P/OS Server systems, these models can act as the server for one or more workstations.

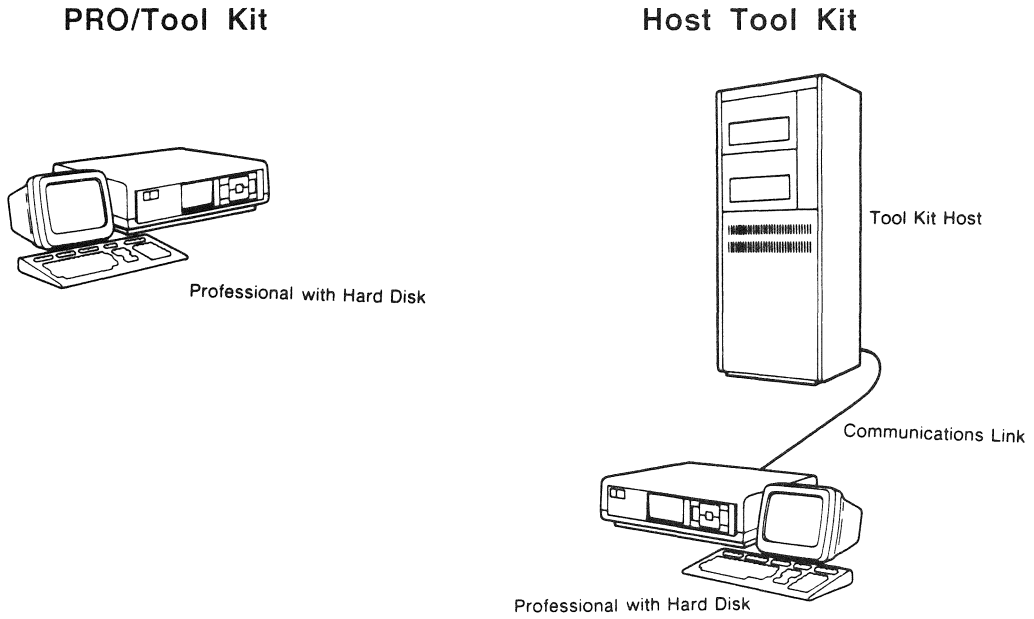


Figure 1-1: The Two Tool Kits

1.2.1 Hardware

For all models, the basic system consists of a system unit, video monitor, and keyboard. Storage media are optional.

The system unit houses a 16-bit PDP-11 CPU, located on the main system board. The base Professional provides 512KB of random access memory. Additional memory can be added in 256KB increments.

The standard video monitor (VR201) has a 12-inch diagonal, low-reflection screen, with tilt adjustment in the back.

The keyboard has four distinct regions: top row function keys, editing keypad, auxiliary keypad, and main array (traditional keyboard). See Figure 1-2.

The Professional has two kinds of storage media: diskettes and hard disk. The standard dual diskette drive provides over 800KB of storage capacity on two 5-1/4" diskettes. The available Winchester hard disks provide a wide range of storage capacities.

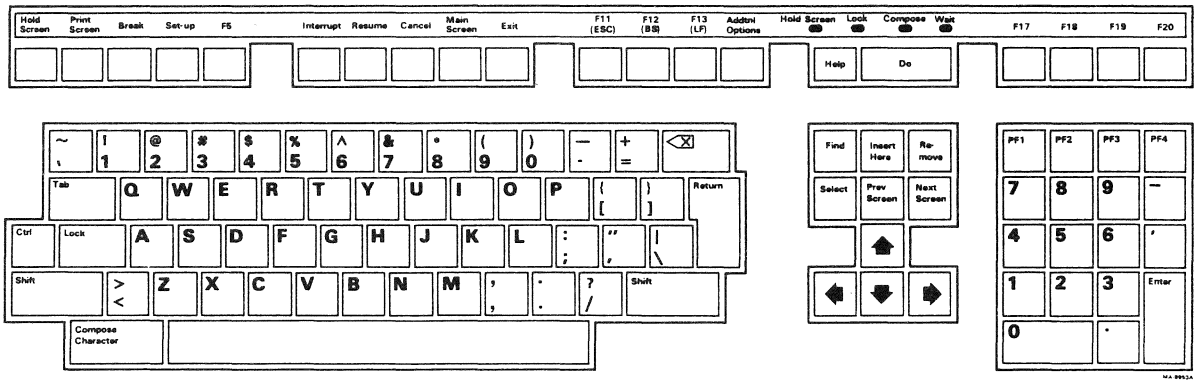


Figure 1-2: Professional Keyboard (U.S./Canada)

1.2.2 P/OS Server

P/OS Server is a component of P/OS that provides the following features:

- Ability to share resources, such as printers and storage media, among several Professionals.
- Access to all P/OS Hard Disk features without the need for a local hard disk.

In a P/OS Server system, one Professional is the server; it contains the hard disk from which the operating system image is loaded. Workstations are connected to the server via DECnet. Each workstation has a DECNA board in an option slot, which supplies the communication capability required by P/OS Server. A typical configuration can consist of one server and four to eight workstations.

The Tool Kit document set uses "P/OS Server" to refer specifically to a P/OS configuration that uses a server, and uses "P/OS stand-alone" to refer specifically to a P/OS configuration without a server. For further information on P/OS Server, see the *P/OS Server User's Guide*.

1.3 DEVELOPMENT LANGUAGES

A MACRO-11 assembler is included with the Tool Kit. You can order other high-level languages separately.

All the high-level languages described in the following sections consist of a compiler and an Object Time System (OTS) Library. The compiler is included with the high-level language product, which is ordered separately. The OTS, however, is supplied with P/OS.

1.3.1 BASIC-PLUS-2

BASIC-PLUS-2 is an extended BASIC compiler. It takes full advantage of the floating point and integer instructions, as well as capabilities of P/OS.

In addition to elementary BASIC language features, BASIC-PLUS-2 provides compile-time directives, structured programming constructs, EXTERNAL statements, language subsets and subset flaggers, exception handling, and many more useful features for application development.

1.3.2 COBOL-81

COBOL-81 is a high-level language for business data processing. Compatible with COBOL-81/RSX, it also shares many features with VAX-11 COBOL; code developed using Professional Tool Kit COBOL-81 can be migrated to VAX-11 COBOL.

COBOL-81 uses an interactive symbolic debugger and includes many DIGITAL extensions to the COBOL language, including advanced screen handling, and file sharing features to enable more than one task to access data simultaneously.

1.3.3 DIBOL

DIBOL is the Tool Kit version of DIGITAL's Business Oriented Language. Similar to COBOL in its use of DATA DIVISION and English-like procedural statements, DIBOL takes extensive advantage of the Professional's architecture.

DIBOL features RMS file support and a resident DIBOL library; it also allows use of system services while maintaining many of the standard DIBOL features. It enables data manipulation, arithmetic expression evaluation, subroutines, structured constructs, table subscripting, record redefinition, external and internal calls to other programs, intertask communication, and random, sequential, and indexed access to files. In addition, DIBOL includes a comprehensive on-line debugging utility, DDT, with which you can quickly isolate and correct programming errors.

1.3.4 FORTRAN-77

FORTRAN-77 is an extended implementation of the ANSI subset FORTRAN-77 standard. Switch-selectable support is provided for user programs based on the previous ANSI FORTRAN standard.

Features include CHARACTER data types, Block IF constructs for the conditional execution of blocks of statements, double precision and complex data types, intrinsic functions, exponentiation forms, format edit descriptors, generic function selection, and virtual array support. FORTRAN-77 also provides access to sequential, relative, and indexed organization files.

The compiler produces direct PDP-11 machine code optimized for execution-time efficiency on a Professional with a floating point adapter.

1.3.5 MACRO-11

The Tool Kit contains the MACRO-11 assembly language processor. Tool Kit MACRO-11 (PMA) is the standard PDP-11 relocatable assembly language processor. MACRO-11 comes with every Tool Kit.

1.3.6 PASCAL

PASCAL is a structured, high-level language that provides a modular, systematic approach to problem solving. It is an extended implementation of PASCAL.

Language features include user-defined and subrange scalar data types, structured variables and constants, loop control statements, functions and procedures, and attributes to facilitate access to P/OS resources. A set of I/O routines support sequential, relative, and indexed files.

Tool Kit PASCAL is an optimizing compiler, generating PDP-11 instructions in the form of binary object modules or MACRO-11 source code.

1.4 DEVELOPMENT TOOLS

1.4.1 Professional Application Builder (PAB)

The Professional Application Builder is the Tool Kit version of the RSX-11M/M-PLUS Task Builder. It is the utility that links your object modules with system software to produce task image files that can be executed on the Professional. PAB is an extremely powerful tool that provides sophisticated programmers with virtually unlimited control over the characteristics of a task image. It is documented briefly in this manual and in detail in the *RSX-11M/M-PLUS and Micro/RSX Task Builder Manual*.

1.4.2 Application Diskette Builder (ADB)

The Application Diskette Builder allows you to easily create a copy of your application on one or more diskettes. The diskettes can then be used to install your application.

ADB is documented in the *Tool Kit Reference Manual*.

1.4.3 Fast Install

Fast Install is a tool that runs on the Professional and allows you to install your application from a hard disk. It is documented in the *Tool Kit Reference Manual*.

1.4.4 Frame Development Tool (FDT)

The Frame Development Tool is used to create frames through which an end user interacts with your application. A frame includes all the information needed to display a menu, or a screen of help or message text. Through a series of forms, you specify the actual text to be displayed in the frame and the information relating to the manner and timing of the frame displays. The Frame Development Tool is documented in the *Tool Kit Reference Manual*.

The Frame Development Tool is used to prepare the frames and to store them in frame files. You must also embed special calls to POSRES services at the proper points in your application code to display the frames. (See Chapter 3.)

1.4.5 On-Line Debugging Tool (ODT)

The On-Line Debugging Tool is special code that you link into your application's task image during the debugging phase. You can use it to control program execution, display and alter the contents of memory locations and registers, search and fill memory, and perform calculations. ODT is provided for debugging MACRO programs, and programs written in high level languages that do not provide their own debuggers. It is documented in the *IAS/RSX-11 ODT Reference Manual* and supplement.

1.4.6 Forms Management System (PRO/FMS-11)

PRO/FMS-11 is the Tool Kit version of FMS-11 for PDP-11 systems. FMS-11 contains tools for constructing forms and callable services for displaying forms during application execution. PRO/FMS-11 is made up of three software components:

- The Form Editor (PROFED) is an editor for creating and modifying video forms.
- The Form Utility (PROFUT) is a utility for creating and maintaining binary form library files.
- The Form Driver (FDV), available as either an object module or resident library, consists of a set of callable subroutines that display forms, perform input and output operations, respond to requests for help, and so forth.

The Tool Kit documentation set includes several manuals devoted to PRO/FMS-11.

1.4.7 Communication Services

Communication Services allow you to perform communication operations on the Professional. They fall into the following categories:

- *Base System Services* are part of the operating system. They include an asynchronous driver (XKDRV) for the communication port, as well as a communication service library (COMLIB). The Base System Services allow you to set up and control a telephone connection for data communication and to handle voice communication if you install the optional Telephone Management System (TMS). These services are documented in the *Tool Kit Reference Manual* and the *P/OS System Reference Manual*.

- *PRO/Communications Services* consists of a set of routines that allow your application to access phone book, communications set-up, and file transfer utilities as well as the Terminal Emulator. These services require that the end user has installed the PRO/Communcations application on the target system. These services are documented in the *Tool Kit Reference Manual* and the *PRO/Communications Manual*.

- *Telephone Management System (TMS) Services* consist of a set of routines that allow your application to control the TMS hardware. These services require that the end user has installed the TMS application and hardware on the target system. These routines are documented in the *Tool Kit Reference Manual* and the *Telephone Management System (TMS) Programmer's Manual*.

1.4.8 CORE Graphics Library (CGL)

The CORE Graphics Library is a general-purpose graphics subroutine package based on the ACM SIGGRAPH CORE Standard, with additional instructions that provide high-level access to the Professional series video bitmap. Features include:

- Automatic mapping of user-defined coordinates to graphics devices

- Lines, curves, polygons, rectangles, solid objects, and graphics text

- Control of styles, textures, and colors

- Multiple user-defined fonts

- Support for graphics plotters

- Easy access from high-level languages
- Device independence and transportability

The CORE Graphics Library is documented in the *CORE Graphics Library Manual*.

1.4.9 PRO/DECnet

PRO/DECnet software allows Professional computers to communicate with other DECnet systems. PRO/DECnet is an end-node-only implementation of the Phase IV Digital Network Architecture. It is compatible with other Phase III and Phase IV DECnet products. The PRO/DECnet software supports:

- Multiple, simultaneous logical links between a Professional and any other Phase III or Phase IV DECnet system
- Task-to-task communication between a Professional and any other Phase III or Phase IV DECnet system
- Resource sharing within a wide-area network
- Various network management and maintenance functions
- Transport facilities that permit programs using RMS-11 V2.0 to access remote files

The Tool Kit documentation set includes a volume devoted to PRO/DECnet.

1.4.10 General Image Display Instruction Set (GIDIS)

GIDIS is a general-purpose graphics subroutine package that provides low-level virtual device access to the Professional video bitmap. Use PRO/GIDIS when speed and compactness are of primary importance, such as with the following software:

- Interactive drawing packages
- Graphics terminal emulators
- Scientific or engineering data displays

- Rapid picture display programs

The preferred application interface for PRO/GIDIS is the GIDIS Call Interface (GIDCAL), which you can use with either MACRO-11 or high-level languages. Also, with MACRO-11 you can use the RSX QIO call.

PRO/GIDIS is documented in the *PRO/GIDIS Manual*. See that manual for a more detailed comparison of CGL and PRO/GIDIS.

1.4.11 File Control Services (FCS-11)

File Control Services is a set of file management routines for use on the RSX-11 family of operating systems. Although P/OS provides a full implementation of FCS, you are urged to always use RMS in new applications. Use FCS only to port applications designed to run on RSX systems when such applications already use FCS.

FCS is described in the *IAS/RSX-11 I/O Operations Reference Manual*, which is included in the RSX-11M or RSX-11M-PLUS documentation set (but not included with the Tool Kit).

1.4.12 Print Services

Print Services consists of a routine that allows your application to print a file, stop, continue, abandon or restart a print job, or obtain printer status. Print Services is documented in the *Tool Kit Reference Manual*.

1.4.13 POSRES User Interface Services Library

POSRES consists of a set of routines that allow your task to have a menu-based user interface consistent with that used by the operating system. These routines open and close frame files; pack, unpack, read, and display menus, help frames, and message frames; invoke the New File and Old File frames; and process function keys.

See Chapter 3 for a description of how to design and implement a menu-based user interface. The POSRES routines themselves are documented in the *Tool Kit Reference Manual*.

1.4.14 POSSUM System Services Library

POSSUM consists of a set of routines that allow your task to manipulate file attributes, directories, volumes, logical names, tasks, regions, and commons. POSSUM is documented in the *P/OS System Reference Manual*.

1.4.15 System Library Routines

System Library Routines provide commonly-used functions for use in your application. The task builder automatically searches the system library file for any referenced routines. See the *P/OS System Reference Manual* and the *IAS/RSX-11 System Library Routines Reference Manual* for details.

1.4.16 PROSE Callable Editor Task (CET)

The PROSE Callable Editor Task allows your application to call PROSE, the text editor supplied with P/OS. PROSE offers facilities for entering and editing text. The end user documentation describes PROSE; the *Tool Kit Reference Manual* documents the callable editor task.

1.4.17 Record Management Services (PRO/RMS-11)

Record Management Services provides an interface between the Professional's file system and your application. All of the Tool Kit high-level languages include support for PRO/RMS-11.

PRO/RMS-11 includes a set of run-time service routines that enable direct, sequential, and multikeyed access to data files. The routines also let your program define, populate, update, and maintain files on direct access devices.

The symbol tables, object module libraries, and overlay descriptor files for task building P/OS applications against the PRO/RMS-11 resident library are included with the Tool Kit, as well as a PRO/RMS-11 Macro library. The Tool Kit documentation set includes several manuals devoted to PRO/RMS-11.

1.4.18 SORT Utility (PRO/SORT)

The SORT Utility is a general-purpose sorting utility that is

callable from your applications. PRO/SORT is documented in the *Tool Kit Reference Manual*.

1.5 STEPS FOR CREATING APPLICATIONS

The application development cycle consists of several steps:

- In the *Design Phase* you make several decisions that will affect all of the other phases. The decisions you make can include your target system configurations, as well as how you will use virtual and physical memory. You also plan the algorithms you will use to implement your application.

- In the *Implementation Phase* you use an editor to create a set of files that specifies the data and code used in your application. These files are normally source code files, command files, form description files, frame files, data files, and so forth.

- In the *Build Phase* you use some of the tools provided to convert your source files into other files that represent an actual working version of your application. These files include executable images, form libraries, converted frame files, and so forth.

- In the *Test Phase* you debug your application tasks, and test an installed version of your application on a Professional. Special debugging tools are provided with the Tool Kit (such as ODT), as well as some of the Tool Kit programming languages.

- In the *Tuning Phase* you can make adjustments to optimize performance and use of resources.

- In the *Distribution Phase* you create a master distribution kit for duplication.

The following two figures illustrate sample development cycles for a typical application. Figure 1-3 shows development on the PRO/Tool Kit; Figure 1-4 shows development on the Host Tool Kit.

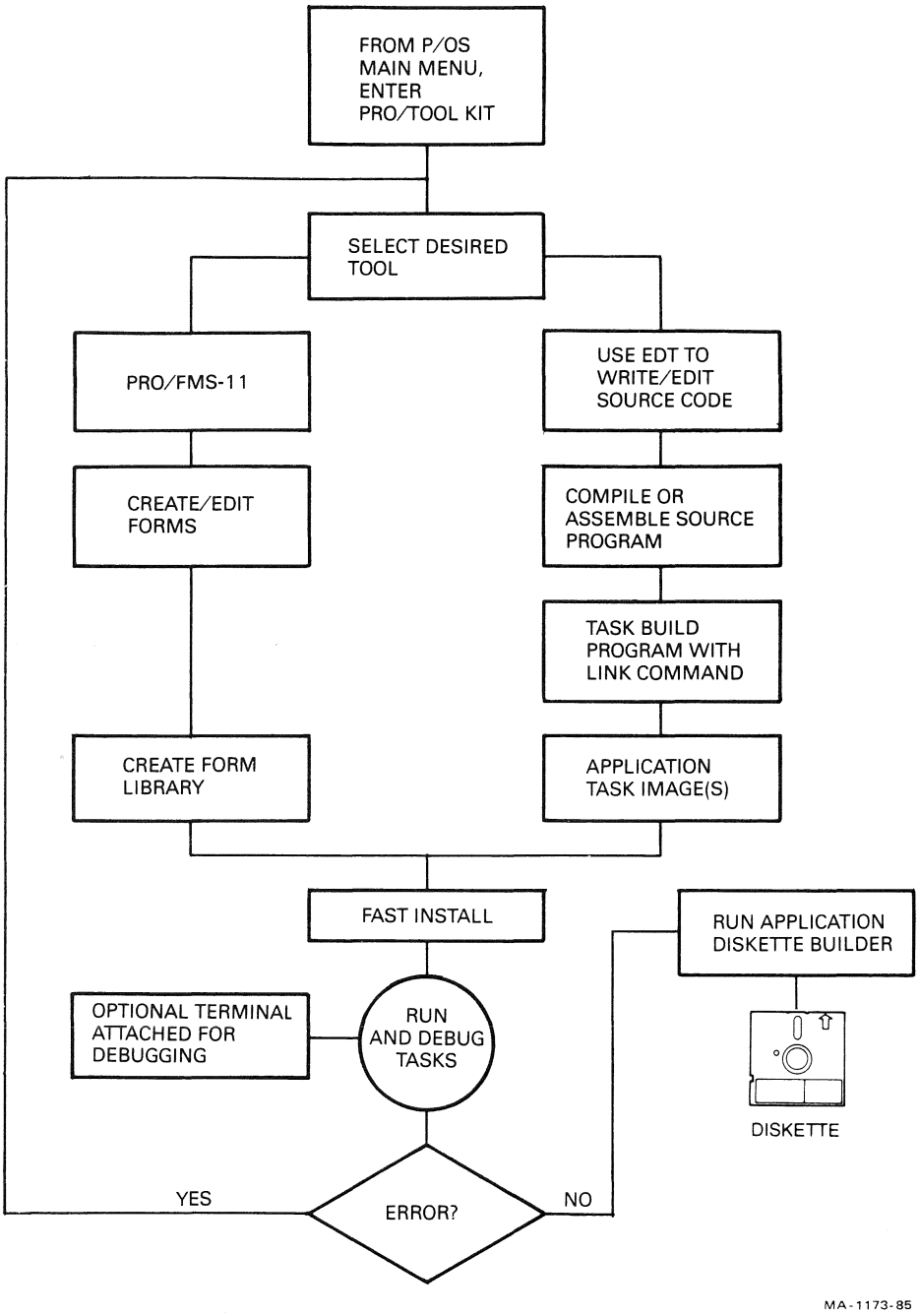


Figure 1-3: PRO/Tool Kit Development Cycle

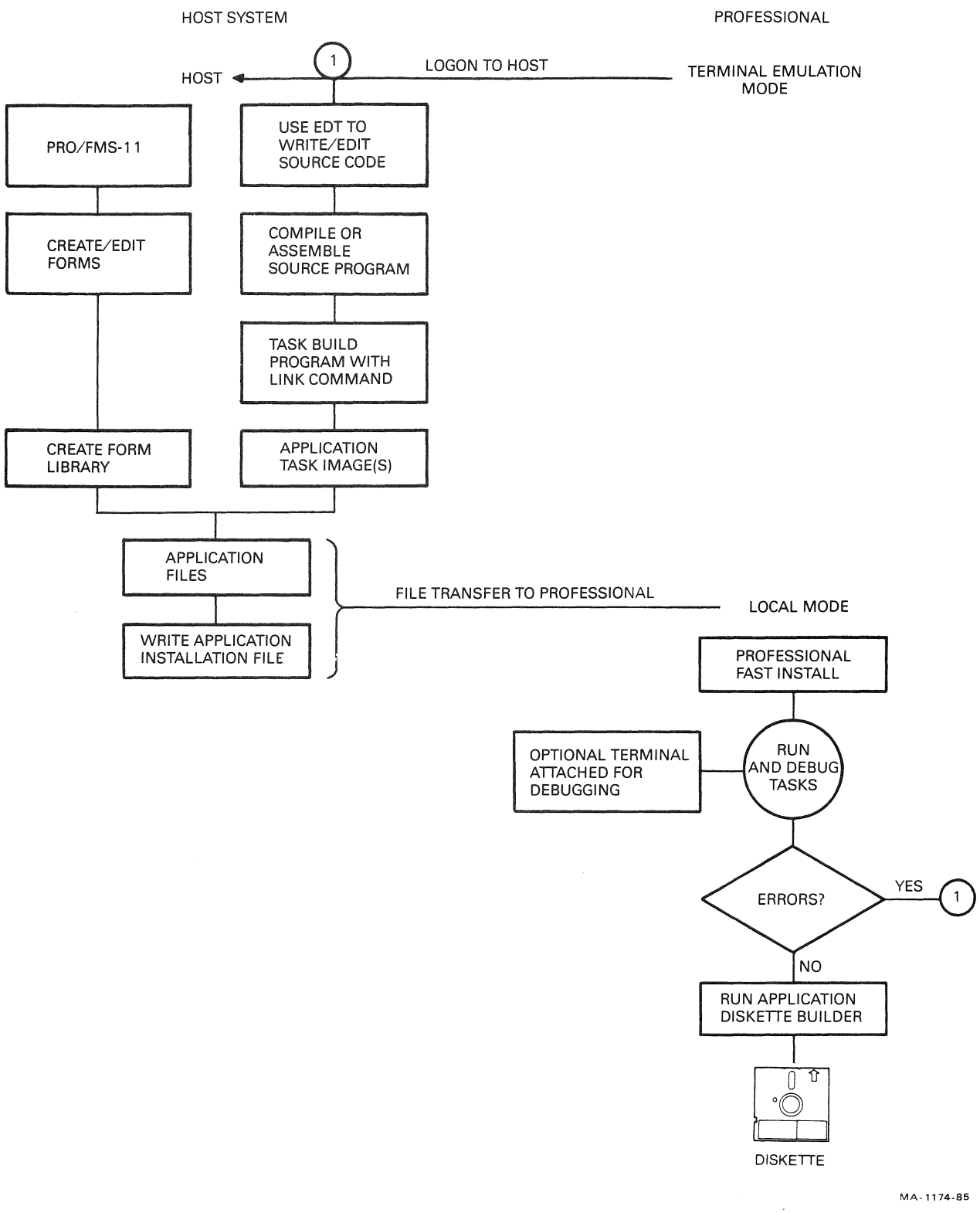


Figure 1-4: Host Tool Kit Development Cycle

CHAPTER 2

DESIGNING THE APPLICATION

This chapter describes several considerations that apply to the design of an application.

2.1 SHARED APPLICATIONS

A *shared application* is an application containing disk-resident components that can be concurrently shared among workstations in a P/OS Server environment. A shared application normally uses less disk space than the equivalent standalone application.

Applications containing shared components have special requirements for installation and removal. The *Tool Kit Reference Manual* describes these requirements in the section on the installation command language.

Other considerations for writing shared applications are:

- File sharing (see the *RSX-11M/M-PLUS RMS-11 User's Guide*)
- Use of read-write commons (see the *P/OS System Reference Manual*)

2.2 VIRTUAL ADDRESS SPACE

User tasks on the Professional are limited to a virtual address space of 64KB. Tasks can map to other regions in memory, but at any given instant only 64KB of memory can be addressed. This places a constraint on the amounts and locations of code and data that a task can contain.

There are several options that you can use, should your task become so large that 64KB is no longer sufficient. These options are:

- Memory-resident overlays
- Clustered resident libraries
- Fast remapping feature
- Disk-resident overlays
- Cooperating tasks
- Shared regions

See the *P/OS System Reference Manual* for details on these options.

2.3 PHYSICAL MEMORY

You must consider not only the amount of virtual address space available to your application, but also the physical memory constraints of the system. It is possible to create an application that runs slowly (or not at all) due to memory contention or deadlocks. These can result from either ignoring the issue of physical memory or overestimating its size.

A base Professional has 512KB of memory. Slightly less than one-half--approximately 240KB--is available for use by the application. Into this memory must fit all tasks, data regions, libraries, and drivers required concurrently by the application. If the application exceeds this amount, checkpointing results.

See the *P/OS System Reference Manual* for details on checkpointing.

2.4 USING P/OS FILES AND FILE SPECIFICATIONS

A file is a collection of data stored on a *volume*. A volume is essentially a physical device containing a storage medium, such as a hard Winchester disk or a diskette.

Within every volume, user files are grouped together in directories, called User File Directories (UFDs). The system maintains a list of UFDs for each volume in a single Master File Directory (MFD), named [0,0]000000.DIR. For example, the UFD [USERFILES] is a file listed in the MFD; it is named [0,0]USERFILES.DIR.

A directory file, such as [0,0]USERFILES.DIR in the preceding example, contains a list of files in the directory ([USERFILES] in this case). This manual uses the term *directory* to indicate the group of files, and *directory file* to indicate the file that points to the group.

To refer to a unique file, directory, or device, you use a *file specification*. A file specification is a character string that consists of one or more fields, separated by punctuation marks, in the following format:

```
nodespec::device:[directory]filename.type;version
```

where:

nodespec	A node specification that indicates the name of a node on your network. See Section 2.4.1.
device	An alphanumeric string that specifies the name of a peripheral device. See Section 2.4.2.
directory	An alphanumeric string that specifies the name of a directory. The maximum length of a directory name is 9 characters. A directory name can be enclosed in either square brackets (as shown above) or within angle brackets (<directory>). See Section 2.4.4.
filename	An alphanumeric string that specifies the name of a file. The maximum length of a filename is 9 characters. See Section 2.4.5.
type	An alphanumeric string that specifies the type of a file. The maximum length of a file type is 3 characters.
version	A numeric string that specifies the version number of a file. See Section 2.4.7.

Note the following:

- File specifications are not case-sensitive. For example, the following specifications are identical:

  DW2:[USERFILES]TEST.DAT;7 dw2:[userfiles]test.dat;7

- You can omit fields from a file specification and let the system or your application provide default values.

- You can substitute logical names for one or more components of a file specification. See Section 2.5 for details.

Examples of various file specifications follow:

```
DW1:                          (device)
DW1:[USERFILES]               (device and directory)
DW1:[USERFILES]TEST           (device, directory, and name)
DW1:[USERFILES]TEST.DAT       (device, directory, name,
                               and type)
DW1:[USERFILES]TEST.DAT;7     (device, directory, name,
                               type, and version)
    [USERFILES]TEST.DAT;7     (directory, name, type,
                               and version)
               TEST.DAT;7     (name, type, and version)
```

The following sections describe each field in a file specification.

2.4.1 Node Specification

If your target Professional is one on which PRO/DECnet is installed, you can operate on files at other nodes on the network by using a node specification. From DCL, note that you can use a node specification only on certain commands; see the *PRO/Tool Kit Command Language and Utilities Manual* for details.

A node specification consists of the name of the node and optional access control information for that node, followed by two colons. The format follows:

```
nodename"access-control"::
```

where:

nodename — Is a 1- to 6-alphanumeric-character name that includes at least 1 alphabetic character. If a node name is an alias that includes associated access control information (a logical node name), you can omit all access control fields, as they will default to the information associated with the alias. For more information on aliases, see the *PRO/DECnet User's Guide*.

access-control — Is information consisting of three position-dependent fields appended to a node name. Specify access control information in the following format:

"userid passwd accnt"::

userid is a string identifying the user at the remote system. For most systems, the userid is the same as the login id.

passwd is an optional string used to specify a password needed to gain access to the remote file system. For most systems, the password is the same as that for logging in.

accnt is an optional string used to specify a billing account number at the remote system. This field is mainly used by TOPS-10 and TOPS-20 systems.

Both the interpretation of the access control fields and the access control mechanism depend on the type of remote system used.

Examples of valid node specifications follow. Fields can be omitted from the right. For example, the node specification QUEBEC"[310,2]" spcifies only a username and not a password or account.

```
YUKON"5,10 LEFT"::
NODE1"RMES"::
NODE4"[7,7] SECRET ACCNT"
```

2.4.1.1 Foreign Files - Files that reside on nodes running operating systems other than RSX or P/OS are referred to as foreign files. Foreign files must use syntax compatible with the systems on which they are located.

When using a foreign format, enclose the portion of the file specification after the node and access control string in quotation marks (""). This directs the system to transmit the file specification to the foreign node without checking its syntax or applying defaults for missing fields. File specifications within quotation marks are not used in determining default values for output files. See the foreign system's documentation for for information on the its format.

The use of wildcards in foreign file specifications is subject to the restrictions of the foreign operating system. See Section 2.4.8 for information on wildcards.

2.4.2 Devices

Each mounted, FILES-11 mass storage device known to P/OS has a logical name supplied by the system. (See Section 2.5 for a description of logical names.) The equivalence value for any logical device name can be one of the following:

- Another logical device name
- Concealed device name
- Physical device name

The following example illustrates how the system translates all three types of logical device names.

Suppose your application opens a file on the device DW001:. On P/OS the following logical name-equivalence pairs exist:

```
DW001: = BIGVOLUME:.
BIGVOLUME: = _DW001:
```

When your application opens the file on DW001:, the system translates DW001: into *another logical device name*--BIGVOLUME:. The system then translates BIGVOLUME: into _DW001:.

Upon reaching a logical device name that begins with an underscore, the system stops performing logical name translations.

Once having performed the final logical name translation, the system strips the underscore and attempts to perform one more translation. In this translation, the system searches for a *concealed device name*. Note that not all logical device names translate into a concealed device name.

In our example, the concealed device name translates into an equivalence value as follows:

```
DW001: = _DW002:[username.]
```

where:

username — Is the root directory associated with the current user's home. The user's home is that user account area containing the user's menus, setup and default information, and user-specific application components.

Upon completion of the concealed device name translation (if any), the system searches for a physical device name. A valid logical device name always ultimately resolves to a physical device name.

In our example, the system uses the DW002: portion of the concealed device name equivalence as the actual physical device name.

2.4.3 Format of Physical and Concealed Device Names

The general format of a physical or concealed device name is as follows:

```
ddnnn:
```

where:

dd — Is a two-character alphabetic string that specifies the device type.

nnn — Is an optional 1- to 3-digit numeric string that specifies one device of a particular device type. This is also called the *unit number*. If you omit the unit number, the system by default uses the unit number 0.

When referring to a device, you should always use a logical device name, not the physical name. This enables your application to be independent of the target system configuration as well as the operating system version.

Many logical device names mimic the ddnnn: form of their equivalent physical names. For example, the logical name DW002: resolves to the physical name DW002:. Some logical device names, such as BIGVOLUME: and SYSDISK:, do not follow the ddnnn: format. You can use either kind in a file specification.

Logical device names provide you with device independence. For example, the logical device name SY000: allows you to refer to the currently active (default) device without regard to the actual physical device name. P/OS handles the assignment and translation of the logical names.

Table 2-2 on page 2-19 lists and describes all the logical device names along with other system-defined logical names.

2.4.3.1 Default Device Names - When a file specification is used, if it contains no device name, P/OS supplies the logical device name SY000: by default.

The end user or the application code can change the default device. When the default device is changed, P/OS reassigns the device name SY000: to the new device.

Note that the end user owns and controls the SY000: logical name assignment. Your application should only reassign SY000: in response to a user query. SY000: need not point to a valid device. Your application should be prepared to handle any errors resulting from attempted accesses to SY000:.

In the absence of any user-supplied device specification, access user data files via SY000:.

To set the default device, do the following:

- Use the P/OS menu system to specify the default device, as described in the *P/OS Hard Disk User's Guide*.
- Use the SET DEFAULT command in DCL. See the *PRO/Tool Kit Command Language and Utilities Manual* for details.
- Call the PROLOG system library routine from an application. The system library (POSSUM) routines are documented in the *P/OS System Reference Manual*.

NOTE

Use PROLOG to change the default directory only under explicit direction from the end user, since the end user owns the right to specify the default device.

2.4.3.2 Referring to Hard Disks - Hard disks that are local to the CPU on which your application is running have the physical device names DW002:, DW003:, and so on. However, you should not normally refer to a physical device name.

In cases in which references to hard disks are context-sensitive, you should use a P/OS system logical device name (not the physical name) to uniquely identify a hard disk. *Context-sensitive* references are those that depend on the current user, the current workstation, or the local device hardware configuration.

For example, a reference to SYSDISK: is context sensitive because it depends on the user running your application--it refers to that particular user's home. Likewise, a reference to LB001: is workstation dependent, and thus context sensitive, because each workstation has a single, unique LB001:.

There are times, however, when your application cannot rely on a context-sensitive reference to a hard disk. To provide the capability of making *context-insensitive* references, P/OS also recognizes a long form for hard disk devices:

```
node$$volumelabel:
```

where:

node	Is the DECnet node name of the system (workstation or server) that controls the disk.
volumelabel	Is the disk's volume label.

The long form is a means of uniquely identifying a particular disk on a P/OS Server system, regardless of the context of your reference. You can access any hard disk on a P/OS Server system using the long form.

We recommend that you use the long form whenever your references to hard disks must be context-insensitive. This ensures compatibility with the widest range of target systems.

Note that the long form remains the same, regardless of whether the access is performed from a workstation or directly on the Server. However, it will change if the system's node name changes. You can maintain a database associating resource names with node names to avoid problems with changing node names.

You can translate device names of the form ddnnn: to obtain the equivalent node$$volumelabel: form. See Section 2.4.3.4 for details.

2.4.3.3 Referring to the User Account Area - SYSDISK: refers to the User Account Area; it is the place where user-specific application components reside. The User Account Area has a structure that is similar to that of a complete hard disk--you refer to SYSDISK: as you would refer to any valid device name. The device logical name DW001: also refers to the User Account Area.

Note that the User Account Area can be located either on a hard disk that is local to the CPU on which your application is running, or it can be located on a hard disk connected to the server in a P/OS Server system.

To distinguish a particular User Account Area on a P/OS Server system, you can use the long form:

```
node$$volumelabel$account:
```

where:

node	Is the DECnet node name of the system (workstation or server) that controls the disk.
volumelabel	Is the associated hard disk.
account	Is the name derived from the user name when the system manager adds an account. This is the name of the User Account Area.

You can translate the special logical name USER$HOME: to obtain the long form of the specification for the current User Account Area.

2.4.3.4 Obtaining Long Forms of Device Names - Suppose you have a file specification that contains a device name (either explicitly or as the result of logical name translation). To obtain the long form of that device name (node$$volumelabel:), do the following:

1. Open the file, or invoke the system directive ACHN$, to assign a logical unit number (LUN) to the device.

2. Using the LUN specified in the previous step, invoke the GLUN$ directive to obtain the device name and unit number.

3. Convert the unit number from binary to octal, and use it to form a three-character ASCII string. Construct the logical device name in the form DDnnn:.

4. Invoke the TLOG$ directive to translate the logical name DDnnn:. The result is the long form of the device name.

2.4.4 Directories

A directory is a named set of files on a disk or diskette. Within a directory, a filename, type, and version number uniquely identify a file. Different files with the same name, type, and version number can exist in other directories.

In a file specification, square brackets ([]) or angle brackets (<>) indicate that the contents are a directory name. Directory names can have the following formats:

- A one- through nine-character alphanumeric string. For example:

  [PROGRAMS] <INVENTORY> [RECIPES]

- A two-part octal number in the format of a user identification code (UIC). Separate the group number from the member number with a comma. For example:

  [0,0] [1,5] [150,13] [240,222]

- A six-character numeric string in UIC format. Omit the comma and specify right-justified (zero-filled) numbers. For example:

  [000000] [001005] [150013] [240222]

Your application can use the POSSUM library routine PRODIR to create and delete directories. (See the *P/OS System Reference Manual*.)

2.4.4.1 Default Directory Name - If a file specification contains no directory name, P/OS provides one by default. A pair of empty square brackets ([]) is an explicit request for the default directory.

The end user owns and controls the default directory name. Your application should change the default directory name only in response to a user query. Note that the default directory name can refer to a nonexistent directory; your application should be prepared to handle any errors resulting from attempted accesses to a nonexistent directory.

In the absence of any user-supplied directory name, your application should allow the system to provide the default directory name.

To set the default directory, do the following:

- Use the P/OS menu system to specify the default, as described in the *P/OS Hard Disk User's Guide*.
- Use the SET DEFAULT command in DCL. See the *PRO/Tool Kit Command Language and Utilities Manual* for details.
- Call the PROLOG system library routine from an application. The system library (POSSUM) routines are documented in the *P/OS System Reference Manual*.

NOTE

Use PROLOG to change the default directory name only under explicit direction from the end user, since the end user owns the right to specify the default directory name.

2.4.4.2 System Directories - Numbered directories and directories that begin with the letters ZZ are reserved for system software.

Also, User Account Areas are implemented in P/OS as concealed devices.

As an example of the use of a concealed device, suppose the system logical name USER$HOME: is equivalent to the following:

```
MYNODE$$PROVOLUME$MYACCT:
```

An access to the directory USER$HOME:[USERFILES] in this case is equivalent to an access to:

```
MYNODE$$PROVOLUME:[MYACCT.USERFILES]
```

2.4.5 Filenames

A filename is a one- to nine-character alphanumeric string that is generally used as a mnemonic name to identify a particular file within a directory. Some valid filenames are:

ACCOUNTS 001005 INDEX3 MAIL

In a file specification, use a period (.) to separate the filename from the file type.

2.4.5.1 Default Filenames - If a file specification contains no filename, P/OS does not supply one by default. Note, however, that your application can supply one.

2.4.6 File Types

A file type is a three-character alphanumeric string that categorizes a file. P/OS uses a set of standard file types to provide useful defaults. For example, the file type TSK indicates that the file is an executable task image.

A semicolon (;) always separates the file type from the version number by P/OS.

Table 2-1 lists the file types used on P/OS.

2.4.6.1 Default File Types - If a file specification contains no file type, P/OS does not supply one by default. However, your application can supply one.

Table 2-1: File Types

Type	Purpose
BAS	BASIC-11 source program
B2S	BASIC-PLUS-2 source program
CBL	Cobol source program
CMD	Indirect command file
COR	SLP file used to correct a source file
DAT	File containing data, as opposed to code
DBL	DIBOL source file
DMP	Dump file create by the File Dump Utility
DOC	Document file
FTN	FORTRAN source program
FLB	Forms library
GID	GIDIS file
HLP	Converted help file
INB	Installation command file
INS	Installation command file
LST	Listing file
MAC	MACRO-11 source program
MAP	Task Builder map file
MLB	Macro library
MNU	Converted menu file
MSG	Converted message file
OBJ	Object module output from assembler or compiler
ODL	File containing Overlay Descriptor Language to be used by the Task Builder
OLB	Object module library
PAS	Pascal source file
PAT	Correction file used by assembler to create patched object module
POB	Patched object module input for PAT utility
SML	File containing system macro library
STB	Symbol definition file created by the Task Builder
SYS	A system file
TMP	A temporary file
TSK	Task image file
TXT	Text file
ULB	Universal library

2.4.7 Version Numbers

A version number is a number that uniquely identifies files that have the same filename and file type. On P/OS, as well as VAX/VMS and Micro/RSX, version numbers are decimal. On RSX-11M/M-PLUS systems, version numbers are octal.

If you specify version number 0 when referring to a file, the system searches for the highest version of the file that exists. (If you create a file and specify 0 as the version number, the system assigns the highest version number.)

If you specify version number -1 when referring to a file, the system searches for the lowest version of the file that exists. (You cannot create a file with version number -1.)

2.4.7.1 Default Version Numbers - When you create a file that does not already exist, it is assigned version number 1 by default. When you create a file that already exists, it is assigned the next highest version number by default.

2.4.8 Wildcards in File Specifications

In addition to the regular defaults for the current device, the current directory, and the most recent version, you can use wildcards in any file specification to set up temporary defaults for every part of the specification except the device name and node information.

Simple wildcarding use the asterisk (*) to replace all or any field in the file specification.

For instance, the following example refers to the most recent versions of all files on any top-level directory (on the default volume) named TEXT.TXT.

```
[*]TEXT.TXT
```

The next example refers to the most recent versions of all files in any subdirectory of any top-level directory (on the default volume) named TEXT.TXT.

```
[*.*]TEXT.TXT
```

Likewise, the * in place of the version number means "all versions." The following example refers to all versions of the file WOM.BAT on the default volume and in the default directory:

```
WOM.BAT;*
```

You can also use * to replace an entire filename or file type. The following example refers to the most recent versions of all files with the type .BAT and any name on the default volume and in the default directory:

```
*.BAT
```

The next example refers to the most recent versions of all files with the name COMMON and any type on the default volume and in the default directory.

```
COMMON.*
```

Several of the utilities provided with the PRO/Tool Kit allow additional forms of wildcarding. See the *PRO/Tool Kit Command Language and Utilities Manual* for details.

2.4.9 File Protection and Volume Protection

Each user has a unique User Identification Code (UIC) that is assigned while setting up an account. A UIC identifies a user to the system.

UICs look like this:

```
[303,005]
```

The format of a UIC is:

```
[ggg,mmm]
```

The ggg indicates the group number, and the mmm is called the member number. Group and member numbers are octal and range from 1 through 376 (octal).

The UIC group value indicates whether or not the user has system file access. (Users with this access are called *privileged* users.) Users having system file access have group numbers from 1 through 10 (octal). Other users have group numbers from 11 through 376 (octal).

When you create a file, you usually own it. A UIC is an attribute of the file, identifying the owner. Each file also has a protection code. A protection code controls who can access a file and in what ways. The file's owner controls this protection code.

The file protection code specifies four categories of users, as well as four kinds of file access that each category of user can have. When you attempt to access a file, the operating system checks your UIC to determine which of the four user categories you belong to. Your ability to access the file is limited to the types of access that the file's protection code grants to those categories.

There are four kinds of users:

SYSTEM — Those having group numbers of 10 or less are SYSTEM users.

OWNER — The user having the same UIC as the file's owner UIC is the OWNER.

GROUP — All users having the same group number as as the file's owner UIC are GROUP users.

WORLD — All other users.

There are also four kinds of access to files:

READ — The user can read, copy, print, or type the file, or if it is a task, run it.

WRITE — The user can add new data to the file by writing to it. Also, the user can change the file's attributes, including the owner UIC and file protection.

EXTEND — The user can change the amount of disk space allocated to the file.

DELETE — The user can delete the file.

In a protection code, the four types of file access are designated by their first letters. The default protection applied to all files on the system that have not been otherwise protected is equivalent to the protection given in SET PROTECTION/DEFAULT command in DCL, or from the P/OS Set Default File Protection set-up option on the menu system.

If you have not set the default protection, the system uses the following protection for all files created:

```
SYSTEM:RWED,OWNER:RWED,GROUP:RWED,WORLD:RWED
```

To gain access to a file, you must satisfy the protection code of both the file to be accessed and the directory in which the file is listed. For example, to write to an existing file, you need at least read access to the UFD and write access to the file itself. To create a new file, you need both write and extend access to the UFD.

References to files located on LB000: or LB001: use the special protection UIC of [377,nnn].

2.5 LOGICAL NAMES

A logical name is a combination of a name (defined either by you or P/OS) and an equivalence value (any part of a file specification). You can use a logical name to refer to all or part of a file specification.

Logical names provide programs with *device* and *file independence*. For example, from within a program you can refer to an input or output file using logical names rather than physical filenames. Then, between invocations of the program, you can change the input and output files simply by associating the logical names with new physical filenames.

You can perform operations on logical names in several ways:

- In PRO/Tool Kit DCL, ASSIGN, DEFINE, DEASSIGN, SHOW LOGICALS, and SHOW ASSIGNMENTS commands allow you to create, delete, and translate logical names. Many other commands accept logical names as arguments.

- From application programs, you can call the PROLOG system-supplied routine. See the section on the POSSUM library in the *P/OS System Reference Manual*.

- Also from application programs, you can directly call the system directives CLOG$, TLOG$, and DLOG$ to respectively create, translate, and delete logical names. See the section on system directives in the *P/OS System Reference Manual*.

Within the strict context of the logical name facility, a logical name and its equivalence name are simply character (byte) strings. The only restriction to logical name strings and equivalence name strings is that neither can exceed 255 decimal bytes.

2.5.1 System-Defined Logical Names

Several logical names are defined and used by P/OS. Your application can refer to P/OS-defined logical names, but it but must not reassign them. P/OS can run unpredictably or stop processing if these names are reassigned.

Table 2-2 shows the system-defined logical names and equivalence values. In the table, an underscore (_) indicates that the system performs no further translation. The description shows in boldface how you use the logical--as a file specification, a device/directory combination, or simply a device.

NOTE

> Logical names that end in a colon can be placed directly in a file specification. On the other hand, a logical name that does not end in a colon must first be translated before using it in a file specification. You can use the TLOG$ directive or the PROLOG callable routine to perform the translation.

For further details on logical names, including logical name tables and operations you can perform, see the *P/OS System Reference Manual*.

Table 2-2: P/OS System Logical Names

Logical Name	Equivalence Value	Comments
APPL$DIR	SYSDISK:[ZZAPnnnnn]	**Device and directory**--See APPL$USER:.
APPL$DST	(varies)	**Device and directory**--See APPL$DST:

Logical Name	Equivalence Value	Comments
APPL$DST:	(varies)	**Device and directory** that is the destination for application files during the current phase of installation. The logical exists only during processing of the installation file during application installation. See the *Tool Kit Reference Manual* for details.
APPL$NETWORK:	(varies)	**Device** containing network components of an application. These components have the /NETWORK qualifier specified in the installation file. See the *Tool Kit Reference Manual* for details.
APPL$HELP	(varies)	**File specification** of your applications's help (.HLP) file.
APPL$MENU	(varies)	**File specification** of your applications's menu (.MNU) file.
APPL$MSG	(varies)	**File specification** of your applications's message (.MSG) file.
APPL$USER:	SYSDISK:[ZZAPnnnnn]	**Device and directory** containing user components of an application. These components have the /USER qualifier specified in the installation file. See the *Tool Kit Reference Manual* for details.
APPL$SRC:	(varies)	**Device and directory** that is the source of the most recently copied file during application installation. The logical exists only during processing of the installation file. See the *Tool Kit Reference Manual* for details.

Logical Name	Equivalence Value	Comments
BIGDISK:	_DW001:	**Device**--see SYSDISK:. Use BIGDISK: for display only.
BIGVOLUME:	_DW001:	**Device**--see SYSDISK:.
CL000:	TI000:	**Device** used for debugger input and output, called the console listing device.
DISKETTE1:	_DZ001:	**Device**--diskette drive 1. Use only for display.
DISKETTE2:	_DZ002:	**Device**--diskette drive 2. Use only for display.
DW001:	BIGVOLUME:	**Device**--see SYSDISK:.
DZ001:	(varies)	**Device** whose equivalence is the volume label of the volume currently mounted in diskette drive 1. It is assigned when the volume is mounted.
DZ002:	(varies)	**Device** whose equivalence is the volume label of the volume currently mounted in diskette drive 2. It is assigned when the volume is mounted.
LB000:	(varies)	**Device** from which your system is booted. It is also the system library device, which contains all system libraries.
LB001:	(varies)	**Device** representing the system library area containing components specific to a particular CPU, such as the system account file.
LDW001:	(varies)	**Device** name that translates into BIGDISK: for English language systems. The equivalence is language dependent.

Logical Name	Equivalence Value	Comments
LDZ001:	(varies)	**Device** name that translates into DISKETTE1: for English language systems. The equivalence is language dependent.
LDZ002:	(varies)	**Device** name that translates into DISKETTE2: for English language systems. The equivalence is language dependent.
LP000:	(varies)	**Device** name of the default printer local to the CPU. Screen dumps from the PRINT SCREEN key are sent to LP000:.
SY000:	(varies)	**Device** which is the user's current default device. Never change this logical name without consent of the user.
SYSDISK:	_DW001:	**Device** representing the user's home. The user's home is where the user-specific application components reside.
TI000:	(varies)	**Device** name of the terminal from which your application is running.
USERDISK:	SYSDISK:	**Device**--see SYSDISK:.
(varies)	_DZ001:	**Device** whose logical name is the volume label of the currently mounted volume. It is assigned when the volume is mounted.
(varies)	_DZ002:	**Device** whose logical name is the volume label of the currently mounted volume. It is assigned when the volume is mounted.

2.6 ACCESSING APPLICATION FILES

Most applications consist of several files: task images, data files, menu files, help files, and so forth. Unless explicitly placed elsewhere, these files exist in an application directory.

On P/OS versions 2.0A and earlier ("older systems"), there is only one application directory: SYSDISK:[ZZAPnnnnn]. The nnnnn is an integer value that the system defines, based on the number of applications currently installed.

On P/OS V3.0 systems and later ("newer systems"), the default application directory is still SYSDISK:[ZZAPnnnnn]. However, there are several additional locations for application components.

The following sections describe how to access application components for all systems.

- **To refer to files supplied by the user:**

  Use the logical device name SY000:. This is valid for all systems.

- **To refer to user-specific files (/USER files) in known directories:**

  Use the logical device name SYSDISK:. Examples are SYSDISK:[ZZSYS]EDTINI.EDT and SYSDISK:[ZZSYS]PROSE.UDK. On older systems, all application files are user-specific, and all application directories are contained on the device whose logical name is SYSDISK:. This logical is valid for all systems. See the *Tool Kit Reference Manual* section on installation command language for details on /USER files.

- **To refer to user-specific files (/USER files) in unknown directories:**

  Use one of the logical names APPL$USER: or APPL$DIR. Both translate to SYSDISK:[ZZAPnnnnn]. On older systems, you must use APPL$DIR (no colon), translating it first. APPL$USER: is valid only on newer systems. An example reference to a user-specific file is APPL$USER:PARAM.DAT. Both logical names are valid on newer systems. See the *Tool Kit Reference Manual* section on installation command language for details on /USER files.

- **To refer to shared components of your application (/NETWORK files):**

  Use the logical name APPL$NETWORK:, available only on newer systems. An example reference to a shared component is APPL$NETWORK:LIBRARY.TSK. See the *Tool Kit Reference Manual* section on installation command language for details on /NETWORK files.

- **To refer to cluster (/CLUSTER files) and system components in known directories:**

  Use the device logical LB000:. You must know the directory containing the file. Do not attempt to refer to a cluster or system file if you do not know the directory that contains it. LB000: is available on all systems. See the *Tool Kit Reference Manual* section on installation command language for details on /CLUSTER files.

- **To obtain the long form specification of the User Account Area:**

  Use the logical name USER$HOME:. This logical resolves to the node$$volumelabel$account: form. It is available only on newer systems. Section 2.4.3.4 describes how to find the long form of a device name for any particular file.

2.6.1 Menu, Help, and Message Files

The installation command file provides an easy way for an application to access menu, help, and message files. If you provide an ASSIGN MENU, ASSIGN HELP, and ASSIGN MESSAGE line in your installation command file, the system will automatically open the specified file whenever you call a User Interface Library (POSRES) routine that uses it. If your program uses additional menu and/or help files, it must explicitly open them.

CHAPTER 3

IMPLEMENTING THE APPLICATION

You implement the application by creating a number of source files. These files include:

- The source language files for the application code
- The frame files for the menus and help that your application uses to interface with the user, as well as any message files
- The command and overlay descriptor files that direct the application build step
- The installation command file that tells P/OS how to install and execute the application

This chapter describes the steps involved in preparing these files.

3.1 CREATING THE APPLICATION DIRECTORY

To test your application on P/OS Hard Disk, you will need a directory to contain the executable files and installation command file. To create the directory, use the DCL CREATE/DIRECTORY command.

NOTE

> The application directory must have the same name as the installation file. For example, if the installation file is PROGRAM.INB, the directory must be named PROGRAM.

If you are working with the PRO/Tool Kit, you can use this directory to contain all of your application files.

3.2 CREATING THE FRAME AND FORM FILES

Use the Frame Development Tool (FDT), described in the *Tool Kit Reference Manual*, or the Forms Editor (FED), described in the *FMS-11/RSX Software Reference Manual*, to create the frame and form files associated with your application. Sketch out your frames and forms before creating them and keep hard-copy descriptions on hand while working on your source code.

3.3 CREATING THE APPLICATION BUILDER FILES

Use any text editor to create the PAB command files and .ODL files for building a task image, as described in Chapter 4. Most of the Tool Kit programming languages have a facility for generating PAB files that you can tailor to your application. Some of the tools, (CGL, POSRES, PRO/FMS-11, for example) require that you make some changes to the PAB files.

3.4 CREATING THE INSTALLATION COMMAND FILE

One application installation command file is required for each Professional target configuration that your application supports. The format and contents are described in the *Tool Kit Reference Manual*.

3.5 CREATING THE SOURCE CODE

Use any text editor to create your source code. If you are using the PRO/Tool Kit, you can use the EDIT command to invoke EDT (the default) or PROSE. If you use PROSE, do not save word wrap/margin settings.

The following sections describe how to call the P/OS services from your source language code and how to design and implement the menu and help interface.

3.6 HOW TO CALL P/OS SERVICES

P/OS routines use the standard PDP-11 R5 calling sequence convention (sometimes called the FORTRAN Calling Sequence Convention). This section provides some general information about the R5 convention. The subsequent sections describe how to call P/OS routines from each high-level language.

This manual and the *Tool Kit Reference Manual* provide a "Format" description for each P/OS routine; this description shows the external routine name followed by a parameter block. The R5 calling sequence convention requires that you pass all parameters by reference. In other words, the parameter block contains only addresses of parameters, not actual data.

The data type and relative position of each parameter must match that expected by the P/OS routine. If a routine doesn't work correctly, check the parameter data types. One of the most common bugs is the specification of a real (floating point) parameter where an integer is required.

Some languages allow you to pass an expression as a reference parameter. The language's run-time library evaluates the expression, stores it in a temporary location, and passes the address of the location. If your language does not support this, read "expression" as "constant or variable."

You can use arrays for multiword parameters. For example, you can use a 2-word integer array for the POSRES status block. You must, however, know how your language numbers arrays. For example, BASIC-PLUS-2 numbers all arrays from zero, while PASCAL allows you to specify your own numbering scheme.

3.6.1 BASIC-PLUS-2

In BASIC-PLUS-2, external subprogram calls do not have to be declared. A call has the format:

```
CALL name BY REF (p1, p2, ..., pn)
```

where:

name — Is the name of the external subprogram.

BY REF — Specifies that the parameters are to be passed by reference.

p1,p2,... — Are actual parameters as described.

Refer to your BASIC-PLUS-2 documentation for detailed information on calling external routines from BASIC-PLUS-2.

Notes:

- To pass an array, include the empty parentheses () in the BASIC-PLUS-2 call.
- BASIC-PLUS-2 does not allow you to pass array elements by reference.
- You can pass a dynamic string variable, using the LEN function to determine its length. For example:

```
CALL name BY REF (..., S$, LEN(S$), ...)
```

3.6.2 COBOL-81

In COBOL-81, external routine calls do not have to be declared. A call has the format:

```
CALL "name" USING p1 p2 ... pn.
```

where:

name Is the name of the external routine.

p1 p2 ... Are actual parameters as described.

Refer to the *Tool Kit COBOL-81 Documentation Supplement* for detailed information on calling P/OS routines from COBOL-81.

3.6.3 DIBOL

In DIBOL, external subroutine calls do not have to be declared. A call has the format:

```
XCALL name (p1, p2, ..., pn)
```

where:

name Is the name of the external subroutine.

p1,p2,... Are actual parameters as described.

Refer to the *Tool Kit DIBOL User's Guide* for detailed information on calling P/OS routines from DIBOL.

3.6.4 FORTRAN-77

In FORTRAN-77, external subroutine calls do not have to be declared. A call has the format:

CALL name (p1, p2, ..., pn)

where:

name — Is the name of the external subroutine.

p1,p2,... — Are actual parameters as described.

Refer to the *Tool Kit FORTRAN-77 Documentation Supplement* for detailed information on calling P/OS routines from FORTRAN.

3.6.5 PASCAL

In PASCAL, an external procedure declaration has the format:

PROCEDURE name (VAR p1; VAR p2; ... VAR pn)**; SEQ11;**

where:

name — Is the name of the external routine.

VAR — Specifies pass by reference.

p1;p2;... — Are formal parameters as described.

SEQ11 — Specifies the PDP-11 R5 calling sequence.

A procedure call has the format:

name (p1, p2, ..., pn)**;**

where:

name — Is the name of the external routine.

p1,p2,... — Are actual parameters that match the formal parameters in the procedure declaration.

Refer to the *Tool Kit PASCAL User's Guide* for detailed information on calling P/OS routines from PASCAL.

Notes:

- You can declare formal parameters with the READONLY attribute so that you can pass constants as actual parameters.
- You can declare formal string (ARRAY [1..n] OF CHAR) parameters with the UNSAFE attribute so that you can pass strings of different lengths as actual parameters.

3.6.6 MACRO-11

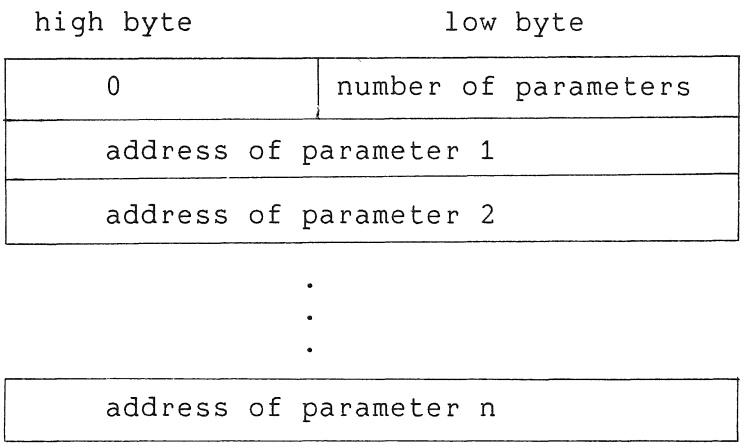
To transfer control to a P/OS routine, use the format:

JSR PC, name

where:

name Is the name (global entry point) of the routine.

General purpose register 5 (R5) contains the address of the parameter block, which has the following format:



When the P/OS routine returns, the contents of registers R0 through R5 are undefined. The stack pointer (SP) is restored to its state at routine entry.

3.7 USING POSRES SERVICES

The P/OS user interface services provide the means by which users can interact with your application in exactly the same manner

that they interact with P/OS. From the user's point of view, the transition from P/OS to application can be just another menu.

The user interface services also allow you to remove all text from your source code, making a single application usable in any number of languages. For example, you can package the same task images with menu, help, and message files in English, French, German, Italian, and so forth.

This section describes how to design and implement a menu-based user interface. To accomplish that, you must be familiar with how a menu interface works from a user's point of view. This chapter assumes that you are familiar with the P/OS user interface. If not, spend some time working with the P/OS menus and help structure before proceeding.

A menu-based user interface allows the user to interact with your application by repeatedly selecting one or more options* from a finite list, called a *menu*. Because all options are visible to the user, it is not necessary to memorize a command language.

In practice, the structure that best matches the user's perception of a menu interface is the multiway tree (hierarchy), where each option on a menu represents a decision and points to another menu. The end points of the tree represent states in which your application has gathered enough information to perform an operation. When the operation is complete, the user repeats the decision-making process, beginning at the main menu or at some other menu in the tree.

Because the menu interface is entirely under program control, the user need not be restricted to downward movement in the menu tree. You can allow movement forward or backward to any other menu in the tree. Backward movement can provide the user with a way to cancel a wrong decision and start over; forward movement can provide an abbreviated or concise way to make decisions.

While a menu is active, some function keys are *trapped* and used as part of the option selection process. Other function keys return control to your application with information about the key that was pressed. This chapter contains some recommendations on how to assign semantic meanings to keys and how to process them.

A menu-based user interface also includes context-sensitive help in the form of help menus, which are similar to control menus, and help text frames, which are simply informative displays.

---

* P/OS end user documentation uses the term *item* rather than *option*.

Help structures can be multiway trees or complex networks of interlocking menus and text frames.

You can associate help structures with menus or with individual options on menus. While a menu is active, the menu interface automatically activates the appropriate help structure whenever the user presses the HELP key. If no menu is active, your application can detect and process requests for help by explicitly activating a help structure. You can even monitor the user's progress through the menu tree and offer help if the user makes repeated errors or seems to be stalled at some point.

A menu-based user interface also provides a way for your application to display context-sensitive messages that announce error conditions, confirm completion of operations, and so forth. Each time it displays a menu, your application can specify up to two lines of messages to appear with the menu at a predefined location. Your application can also send a message to the View Message/Status service on the P/OS Main Menu.

The user interface services consist of:

- **The Frame Development Tool**

  The Frame Development Tool (FDT) is a utility program (you may prefer to think of it as an editor) that allows you to create menu, help, and message frames, to store frames in files, and to convert the files into a format that can be read by POSRES User Interface Services. FDT is described in detail in the *Tool Kit Reference Manual*.

- **The POSRES User Interface Library**

  POSRES is a P/OS clustered resident library containing a set of callable routines that manage menus and help structures (including frames stored in files) and process function keys. This chapter provides a general description of how to use the POSRES routines. A detailed description of each routine can be found in the *Tool Kit Reference Manual*.

Figure 3-1 shows the relationship between FDT and POSRES. FDT allows you to create a frame (in this case, a single-choice menu) by filling in forms. The frame description is then stored in a file and installed with other application files. At run-time, your application uses POSRES routines to open the frame file, retrieve the frame, and activate it.

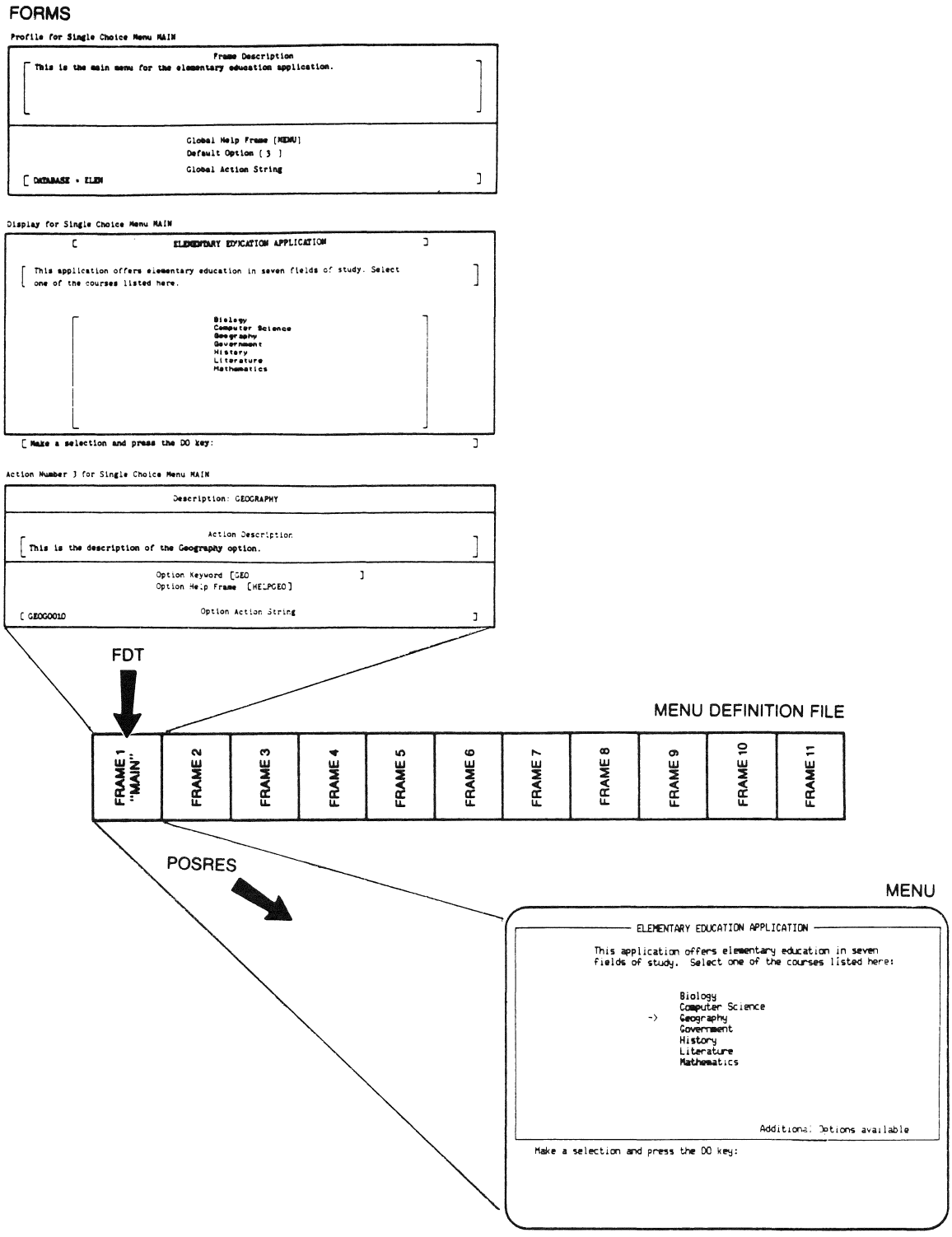


Figure 3-1: User Interface Tools

3.7.1 Designing a Menu Structure

The primary goals in designing a menu tree are consistency and friendliness. The user should never feel lost or trapped. You should always provide a way to back out of a wrong decision.

Convenience is also very important. On any menu, include options that are related by function (rather than program logic) so that common operations can be completed with minimal switching of menus. If necessary, put the same option on more than one menu.

3.7.1.1 Format of a Menu - From the user's point of view, there are three types of menus: single-choice menus, multiple-choice menus, and help menus. They all have roughly the same fields. Figure 3-2 shows a single-choice menu with the fields pointed out.

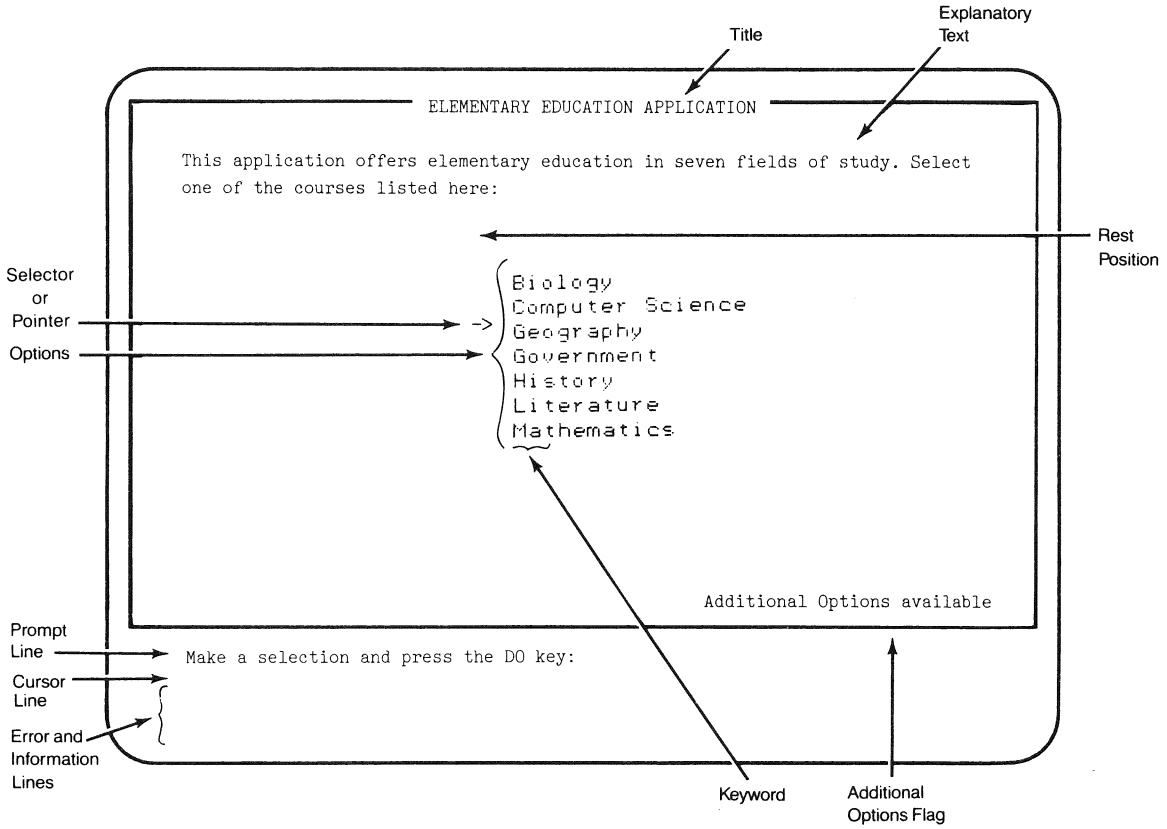


Figure 3-2: Single-Choice Menu

On a single-choice menu, part of each option appears in boldface to indicate which characters form an unambiguous choice. The boldface characters are called the *keyword* for the option.

3.7.1.2 Single-Choice Menus - There are actually two kinds of single-choice menu: static and dynamic (described in Section 3.7.2). The difference is relevant only to programmers; from the user's point of view, there is only one kind.

When the menu is displayed, the selector begins at the default option or, if none was specified, at the rest position. The user selects an option from a list of up to 12 options. There are two ways to select an option:

- The user positions the selector at the desired option by pressing UP ARROW and DOWN ARROW keys, then presses the DO key.
- The user positions the selector at the desired option by typing keyword characters, then presses the DO key. The selector moves only when the characters identify a unique choice.

Single-choice menus are normally used to allow the user to choose a single function.

3.7.1.3 Multiple-Choice Menus - On a multiple-choice menu, the user selects one or more options from a list. The list can be any length, possibly covering several screens. Figure 3-3 shows a multiple-choice menu.

The user selects options by repeatedly positioning the selector (with UP ARROW and DOWN ARROW) and pressing the SELECT key. An arrow remains next to each selected option. The user can cancel a selection by pressing the SELECT key again, or can cancel all selections (start over) by pressing the CANCEL key.

If the option list covers more than one screen, the user can move forward or backward in the option list by pressing the PREV SCREEN and NEXT SCREEN keys. The DO key signals that the selection process is complete.

Multiple-choice menus are normally used for objects (data), rather than commands. A common example is a list of files or directories.

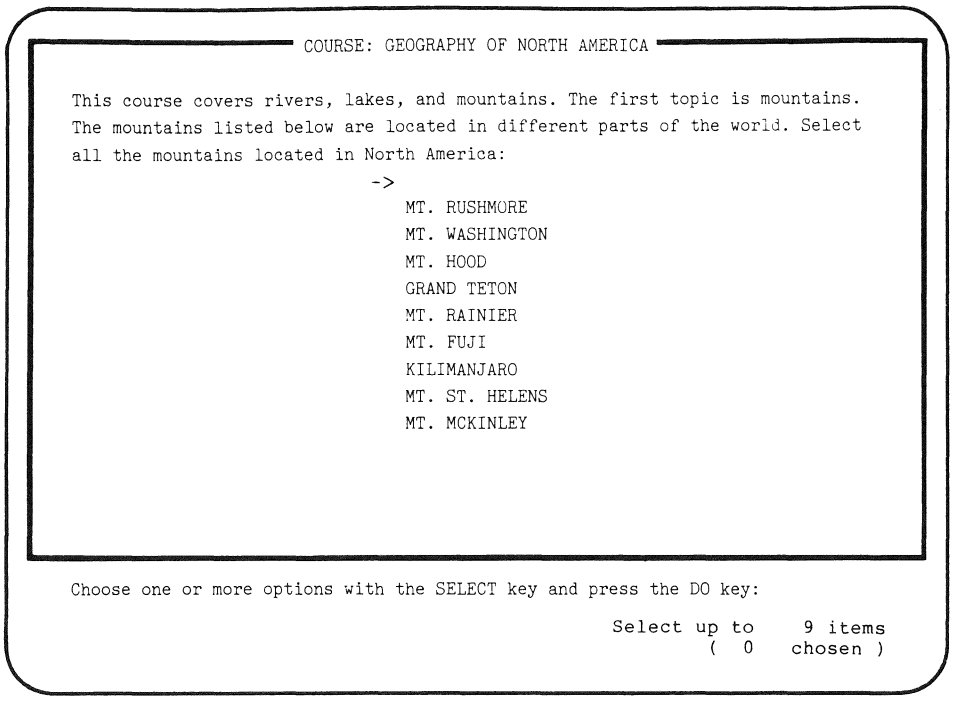


Figure 3-3: Multiple-Choice Menu

3.7.1.4 Key Processing in Menus - This section describes key processing in menu option selection. (All key codes and labels are supplied in the *Tool Kit Reference Manual*.)

- Keyboard keys are accepted if they match an option keyword. Otherwise, the keyboard bell rings and selection continues. The DELETE key deletes the previously typed character.

- The ADDTNL OPTIONS key returns control to your application if your menu display call specified Additional Options. Otherwise, the keyboard bell rings and option selection continues.

- The EXIT key returns control to your application. Whenever your application receives the EXIT key code, you should return to the next highest menu in your menu tree hierarchy.

- UP ARROW and DOWN ARROW move the selector up and down. If the user tries to move the selector out of range, the keyboard bell rings and selection continues.

- The CANCEL key moves the selector to the rest position. On a multiple-choice menu, it cancels all selections.

- The DO key *chooses* any selected options. That is, on a single-choice menu, the DO key selects one option and returns it to your application. On a multiple-choice menu, the DO key selects the current option, and returns it and all selected options to your application. If no option has been selected and the pointer is at the rest position, the DO key rings the keyboard bell.

- The HELP key activates a help structure as described in Section 3.7.4.3.

- The HOLD SCREEN key functions normally.

- The INTERRUPT key does not by itself return control to your application. If followed by the DO key, the system attempts to abort your application. Otherwise, option selection continues as usual.

  Note that your application can specify the SREX$ directive to trap abort attempts. See the *P/OS System Reference Manual* for details.

- The PRINT SCREEN key functions normally.

- The SELECT key selects an option or cancels a selection on a multiple-choice menu, without returning control to your application. However, on a single-choice menu, the SELECT key only returns control to your application.

- Other function keys either ring the keyboard bell or terminate the menu display and return control to the executing task with a numeric code to identify which key was pressed. In general, you should handle invalid keystrokes by redisplaying the same menu with a message such as "You pressed an invalid key; try again."

3.7.1.5 Action Strings - Action strings are character strings that are stored in a menu definition but not displayed; POSRES returns them to your program with specific calls. Their purpose is to associate menus and options with data usable by your task. For example, you can associate menu and options with other menus, tasks, subroutines, or callable services.

There are two types of action strings:

- **Global action strings**

  A global action string associates data with a single-choice menu. Your application can obtain and use the global action string each time it reads a new menu from the menu file.

- **Option action strings**

  Option action strings associate data with options on a single-choice menu. A successful option selection returns an option action string to your application.

3.7.1.6 Option Keywords - On a single-choice menu, you should designate part of each option as the keyword. The user can select the option by typing keyword characters, rather than by pressing ARROW keys.

The keyword must be at least one character. It can be any contiguous substring of an option or the entire option. For example, you could use any of the following:

Enter Accounts **Payable**

Enter **Accounts** Payable

Enter Accounts Payable

A keyword should be unique within the menu. No keyword can be a substring of another keyword. For example, the string "1" is a substring of the string "10".

Verbs are usually the most appropriate keyword. If no keyword is suitable, use numbers. (This is not recommended.) Examples follow:

1 Sales Report: Area One
2 Sales Report: Area Two
3 Sales Report: Area Three

3.7.2 Implementing a Menu Structure

From a developer's point of view, there are two kinds of menus:

- **Static menus**

  Static menus are created with FDT and stored in frame files. The system can generally display a static menu more quickly that it can display an equivalent dynamic menu.

- **Dynamic menus**

  Dynamic menus are created by your application at run-time. In general, dynamic menus are intended for manipulating data.

NOTE

Only single-choice menus can be static or dynamic. Multiple-choice menus are always dynamic.

POSRES uses three buffers for temporary storage of menu frames--one each for static single-choice menus, dynamic single-choice menus, and multiple-choice menus. You allocate memory for these buffers when you build your program (see Section 3.7.7). Because there is only one buffer of each type, you can have only one menu of each type in memory at one time.

3.7.2.1 Displaying Menus - All of the routines for displaying menus accept a parameter that specifies whether to display the Additional Options message and return control if the user presses the ADDTNL OPTIONS key. P/OS generally uses Additional Options menus for services beyond those offered on the current menu.

Static Single-Choice Menus - Static single-choice menus are created with FDT and stored in menu frame files. There are two ways to open a menu frame file:

- The ASSIGN MENU command in your installation command file (see the *Tool Kit Reference Manual*) opens a menu file at run-time.

- The Open Menu File (MFILE) routine explicitly opens menu files. To use it, however, you should provide a complete file specification for the menu frame file. Use one of the P/OS system logical names described in Section 2.6 to fully qualify the file specification.

To display static single-choice menus, use the following POSRES routines:

- **Read Menu Frame (MFRAME)**

  MFRAME reads a specified menu from the menu file into the static buffer.

- **Display Single-Choice Menu (MENU)**

  MENU displays the menu in the static buffer and processes user keystrokes (as described in Section 3.7.1.4). You can specify up to two message lines to appear on the menu.

Dynamic Menus - Dynamic menus are created by your application at run-time. To create a dynamic menu, you must first clear the appropriate buffer (dynamic single-choice or multiple-choice) and then pack it. The POSRES menu packing routines are:

- **Pack Dynamic Single-Choice Menu (DPACK)**

  DPACK packs the dynamic single-choice buffer with a new menu.

- **Pack Multiple-Choice Menu (MPACK)**

  MPACK packs the dynamic multiple-choice buffer with a new menu.

Both routines accept a parameter that clears the buffer. They also accept any number of field parameter groups, each of which specifies the contents of a menu field (see Figure 3-2). Thus, you can pack a menu buffer with a single routine call or several calls.

You also can use the contents of the static buffer to create a dynamic menu. The POSRES menu unpacking routine is:

- **Unpack Menu Buffer (MUNPK)**

  MUNPK unpacks the menu in the static buffer so that its contents can be modified and reused as a dynamic single-choice or multiple-choice menu.

Once you have packed the dynamic buffer, display the menu by calling one of the display routines:

- **Display Dynamic Menu (DMENU)**

- **Display Multiple-Choice Menu (MMENU)**

These routines display the menu in the appropriate buffer and process user keystrokes (as described in Section 3.7.1.4). You can specify up to two message lines to appear on the menu. The MMENU routine includes parameters for setting the maximum number of options the user can select and for receiving the responses.

3.7.2.2 Programming with Menus - When control returns to your application from a menu display routine, the menu remains visible on the screen. Examine the first word of the status block to see what happened.

If it contains +1, option selection was successful. Either the second word of the status block, or the response array, contains the ordinal number of the selected option. You can branch on that value, or use the option action string (if defined).

If the first status word contains -14, the user pressed a function key other than DO; the second word contains a function key code. (All key codes are supplied in the *Tool Kit Reference Manual*.) If your menu display call specified Additional Options, be sure to check for 14 in the second status block word.

Your application must determine which function keys are valid and which are not in the context of the current menu. Section 3.7.1.4 provides some suggestions on how to process function keys. If the user pressed an invalid key, redisplay the same menu with a helpful message.

Any other value indicates that an error has occurred. (All possible status values are supplied in the *Tool Kit Reference Manual*.)

When processing is complete, use the Close Menu File (MCLOSE) routine to close the menu frame file (if open) and exit. Otherwise, the file remains open until your task exits.

3.7.2.3 File Specification Routines - POSRES provides two special user interface routines for working with files.

New Filename (NEWFIL) - The New Filename routine activates the P/OS New File Specification form shown in Figure 3-4 and returns a full file specification in the form:

```
dev:[directory]filename.typ;version
```

The user can fill in the form and press DO, or can press ADDTNL OPTIONS. POSRES automatically displays an Additional Options menu that allows the user to specify a new file type, specify a different directory, or enter an extended file specification.

NOTE

When control returns to your application after execution of the NEWFIL routine, reset menu and help contexts.

If your installation command file contains ASSIGN MENU or ASSIGN HELP commands, the next POSRES call after control returns from NEWFIL automatically reopens those files. If not, do the following:

1. Open the menu file explicitly with MFILE.
2. Reset the menu frame with MFRAME.
3. Open the help file explicitly with HFILE.
4. Reset the help frame with HFRAME.

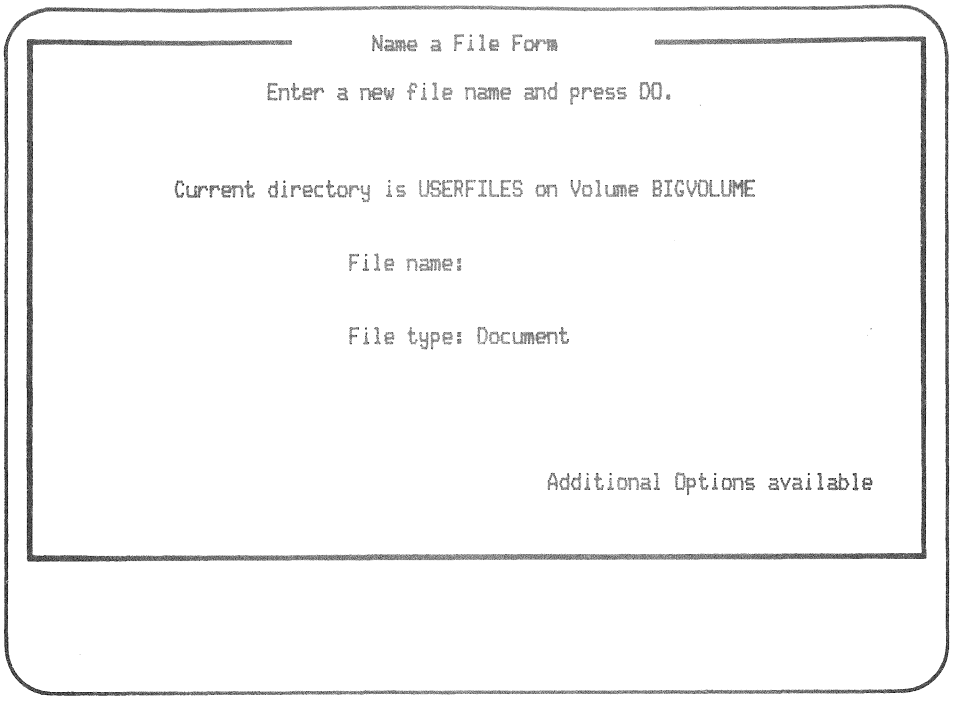


Figure 3-4: Name a File Form

Old Filename (OLDFIL) - The Old Filename routine activates the P/OS File Selection Menu and returns the full specifications of one or more selected files. You can supply a wildcard string to specify a subset of the files in the user's current directory and can specify the maximum number of selections. For example, you can use the default wildcard specification (*.*) by supplying a zero-length string. That displays the latest versions of all files in the user's current directory.

Refer to Section 2.4.8, or your host system documentation, for wildcard syntax.

An Additional Options menu allows the user to choose a different directory or volume, enter an extended filename, show all versions of the files, or show only the latest versions of files. The last two options, however, work only when the application uses the default wildcard specification or when the application passes a wildcard specification that specifies an asterisk (*) for the version number. Otherwise, when the user selects "Show all versions" or "Show latest version," the same file selection menu will redisplay.

NOTE

When control returns to your application, you should reset menu and help contexts.

If your installation command file contains ASSIGN MENU or ASSIGN HELP commands, the next POSRES call after control returns from OLDFIL automatically reopens those files. If not, do the following:

1. Open the menu file explicitly with MFILE.

2. Reset the menu frame with MFRAME.

3. Open the help file explicitly with HFILE.

4. Reset the help frame with HFRAME.

3.7.3 Designing a Help Structure

The primary goal in designing a help structure is to guide the user smoothly through the menu tree. There should be no point in the menu tree where the HELP key does not display help for the user.

Help structures consist of help menus and help frames. Help menus do not provide information; they simply guide the user to the appropriate help text frame or another help menu. Help text frames provide information about your application.

Although you can assume that the user is familiar enough with the menu system to select an option from a menu, you should always state which keys are expected, for example: "Make a selection and press DO," or "Press NEXT SCREEN to continue."

3.7.3.1 Help Menus - On a help menu, the end user chooses one from a list of up to 12 options. Figure 3-5 shows a help menu. The format and option selection process are similar to a single-choice menu.

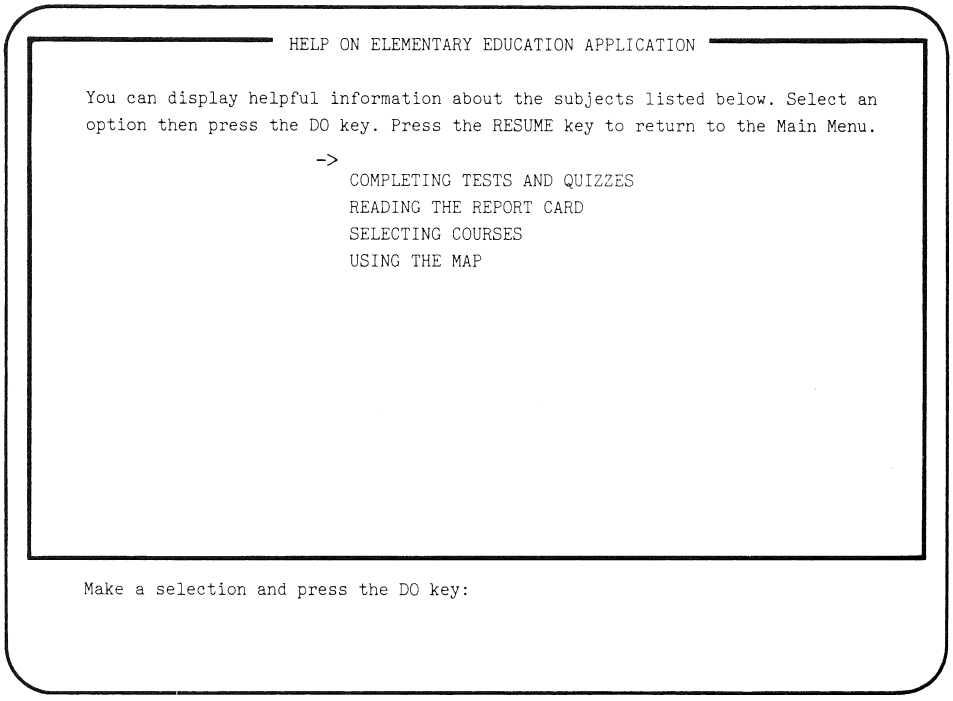


Figure 3-5: Help Menu

3.7.3.2 Key Processing in Help Menus - This section describes key processing in help menus. (All key codes and labels are supplied in the *Tool Kit Reference Manual*.)

On a help menu, option selection keys are processed as follows:

- Main keyboard keys are accepted if they match an option keyword. Otherwise, the keyboard bell rings and selection continues. The DELETE key deletes the previously-typed character.

- UP ARROW and DOWN ARROW move the selector up and down. If the user tries to move the selector out of range, the keyboard bell rings and selection continues.

- The CANCEL key moves the selector to the default option if defined, otherwise to the rest position.

- The DO key displays the frame associated with the selected option if option selection was successful. Otherwise, it rings the keyboard bell.

- The HELP key rings the keyboard bell. In order to continue through the help structure, the user must select an option.

- The HOLD SCREEN key functions normally.

- The INTERRUPT key does not by itself return control to your application. If followed by the DO key, the system attempts to abort your application. Otherwise, option selection continues as usual.

  Note that your application can specify the SREX$ directive to trap abort attempts. See the *P/OS System Reference Manual* for details.

- The PREV SCREEN key displays the previous help frame, if defined. Otherwise it rings the keyboard bell.

- The PRINT SCREEN key functions normally.

- The RESUME key returns control to the menu from which the user entered the help structure or, if the help menu was activated directly by your task, returns control to it.

- Invalid keys ring the keyboard bell but do not return control to your application.

3.7.3.3 Help Text Frames - Help text frames can consist of a full frame (16 lines of text), the top half of the screen (eight lines), or the bottom half of the screen (eight lines). Figure 3-6 shows a help text frame in the bottom half of the screen.

When the user pressed the HELP key, the selector was positioned on the option "Reading the Report Card" in Figure 3-5.

When displayed implicitly by POSRES, half-screen frames do not alter the remainder of the screen. For example, in Figure 3-6, part of the menu remains visible. If, however, your task calls HELP directly to display a half-screen frame, POSRES clears the entire screen.

You should use half-screen frames for menu frame pointers whenever the size of the help text permits. Choose top or bottom in order to allow relevant areas of the previous frame to remain visible. You can design the help structure so that top and bottom frames alternate, allowing the user to see two help frames at a time.

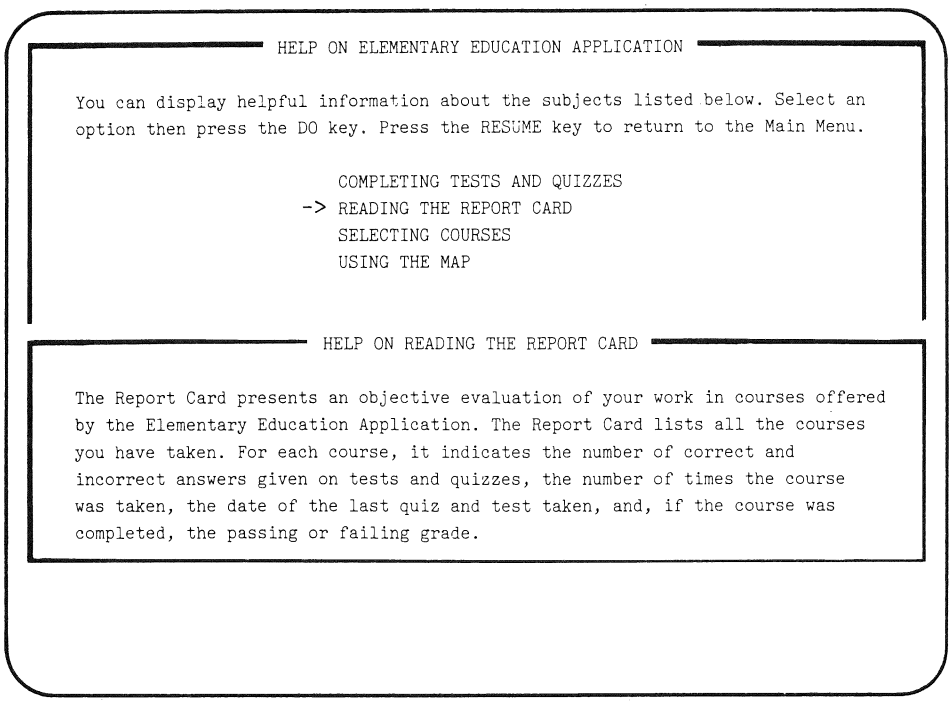


Figure 3-6: Help Text Frame

3.7.3.4 Key Processing in Help Text Frames - While a help text frame is active, only the following keys are processed:

- The HELP key and the NEXT SCREEN key display the next help frame, if one was defined when the frame was created. Otherwise, the keyboard bell rings.

- The PREV SCREEN key returns to the previous help frame, if the frame is defined. Otherwise, the keyboard bell rings.
- The NEXT SCREEN key displays the next help frame, if the frame is defined. Otherwise, the keyboard bell rings.
- The RESUME key returns control to the menu from which the user entered the help structure or, if the help menu was activated directly by your task, returns control to the task.

3.7.3.5 A Sample Help Structure - Figure 3-7 shows part of the system's Main Menu help structure in diagram form.

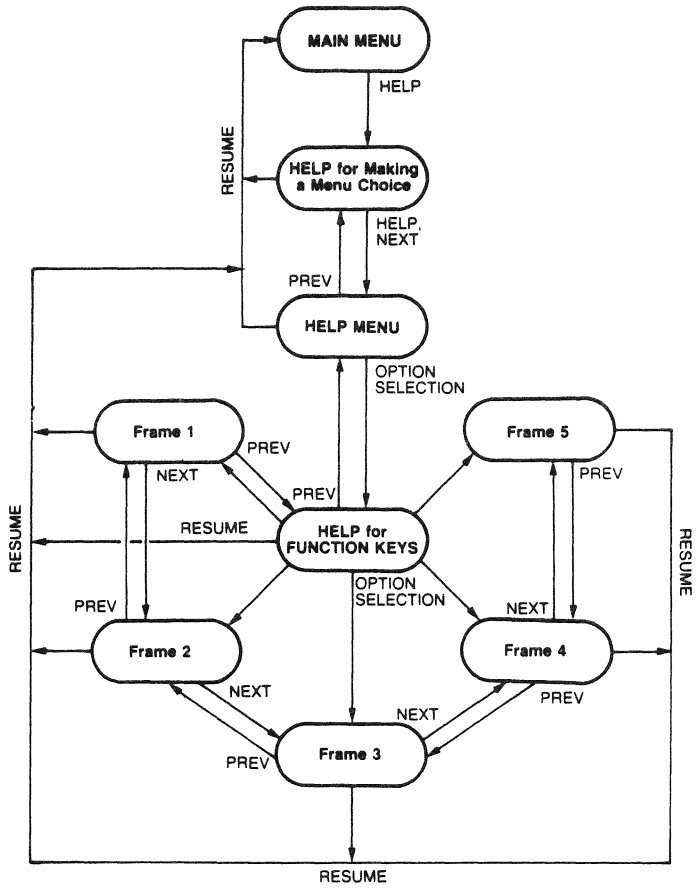


Figure 3-7: P/OS Main Menu Help Structure (Partial)

3.7.4 Implementing a Help Structure

Help structures are made up of frames and frame pointers. A frame pointer is data that specifies a help menu or help frame by name. For static menus, you specify the frame pointers on the FDT Profile and Action forms. For dynamic menus, you specify frame pointers in the parameters passed to the POSRES routines that create the menu.

All menus contain a global frame pointer. When the user presses the HELP key with the pointer in the rest position, POSRES activates the help structure and displays the frame specified by the global frame pointer.

Single-choice menus (static or dynamic) contain a frame pointer for each option on the menu. When the user presses the HELP key with the pointer on an option, POSRES activates the help structure and displays the frame specified by the option frame pointer.

POSRES also maintains a default help frame pointer for use when no menu is active or when the current menu does not define a help frame. Your application can explicitly activate a help structure and specify a new default help frame pointer.

3.7.4.1 Opening Help Files - There are two ways to open a help frame file:

- The ASSIGN HELP command in your installation command file opens a help file at run-time. (See the *Tool Kit Reference Manual* for details on the installation command file.)

- The Open Help File (HFILE) routine explicitly opens a help file. To use it, however, you should provide a complete file specification for the help frame file. Use one of the P/OS system logical names described in Section 2.6 to fully qualify the file specification.

POSRES assumes that all frames used by the current help structure are in the most recently opened help file.

3.7.4.2 Setting the Default Help Frame - There are several ways to set the default help frame:

- The ASSIGN HELP command in your installation command file specifies the default help frame. (See the *Tool Kit Reference Manual* for details on the installation command file.)

- The Open Help File (HFILE) routine specifies a default help frame. This takes precedence over a default help frame specified in your installation command file.

- The Specify Help Frame (HFRAME) routine specifies the default help frame. This takes precedence over a default help frame specified in your installation command file.

- Each time the user presses the DO key to select a menu option, that option's frame pointer, if defined, becomes the default. This automatically maintains the help context.

3.7.4.3 Activating the Help Structure - While a menu is active, POSRES automatically activates the help structure (using the current menu's frame pointers) whenever the user presses the HELP key. This help processing is entirely transparent to your application. POSRES uses the current menu's frame pointers in the following order of precedence:

- The option help frame (single-choice menus only).

- The global help frame.

- The default help frame.

- If there is no default help frame, POSRES simply refreshes the current frame and continues.

While no menu is active, you can activate the help structure by calling the Display Help Frame (HELP) routine. POSRES uses frame pointers in the following order of precedence:

- The help frame specified in the HELP call.

- The default help frame.

- If there is no default help frame, an error occurs.

3.7.5 Message Files and Services

Like help frame files, message frame files are created with FDT. Message frames can contain up to 21 lines of text. By removing the string constants from your task and placing them into a message frame file, you can reclaim a significant amount of virtual memory.

Unlike help frame files, POSRES provides no routines to explicitly display message frames. It does, however, provide a way to extract message frames so that you can use them as parameters to other routines, such as the optional text lines on menus and on special message-service routines.

For example, the Installation Verification Program used by the Tool Kit languages uses that technique for its geography test answers. Figure 3-8 shows the same multiple-choice menu as Figure 3-3, with a message informing the user of an incorrect selection. This was accomplished by extracting message frames and using them to specify the optional text lines in the POSRES calls that created the menu.

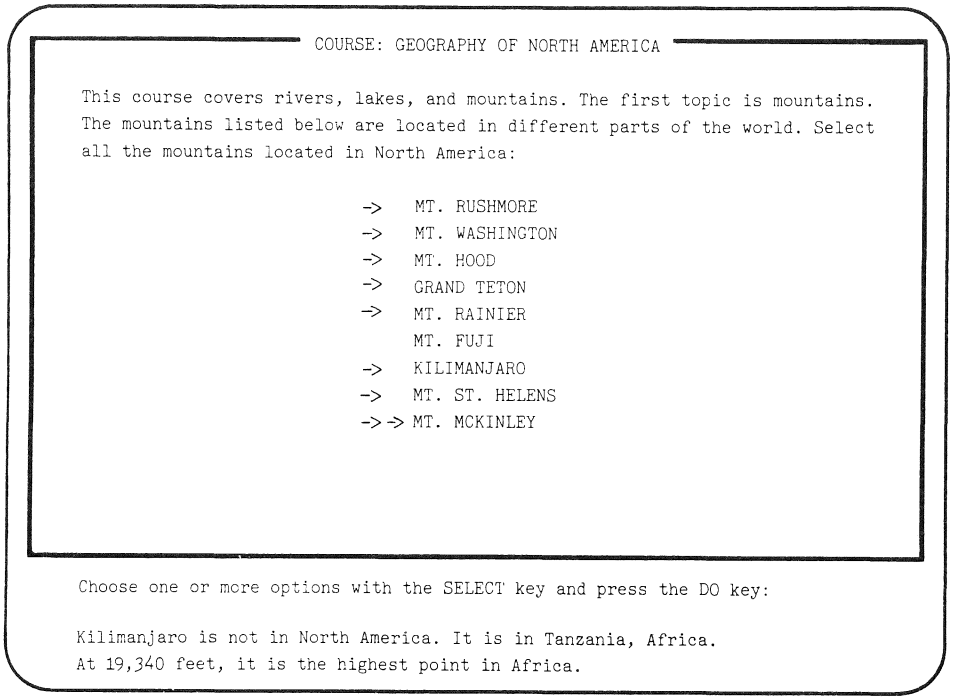


Figure 3-8: Message Frame

POSRES includes the following message service routines (all are fully described in the *Tool Kit Reference Manual*):

- **Assign Message File**

  The ASSIGN MESSAGE command in your installation command file opens a message file at run-time.

- **Fatal Error (FATLER)**

  This routine provides a consistent way to inform the user of a fatal error condition. It blanks line 22; displays the message "Application error. Press RESUME to return to Main Menu." on line 23; and displays user-supplied text on line 24. That text can be a message about why the application failed and where to look for recovery information.

- **Read Message (RDMSG)**

  This routine reads a message from a specified message file into a buffer. You do not have to explicitly open the message file; POSRES opens it for you each time you call RDMSG. If either the file or the frame identifier specified in the RDMSG call cannot not be opened or located, POSRES fills the buffer parameter with the message "Can't access <filename> or can't find frame <frameid>."

- **Send Message to Message/Status Display (MSGBRD)**

  This routine sends a message to the P/OS Message/Status Display, which can be viewed by selecting the "View Message/Status" option on the P/OS Main Menu. The Main Menu shows how many unread messages have been queued. When the user selects "View Message/Status," P/OS displays the messages in the order in which they arrived.

NOTE

The MSGBRD routine is not actually in POSRES; it is in the system library (SYSLIB). If you use the MSGBRD routine, you must also edit your Application Builder command file as described in Section 3.7.7.

3.7.6 Function Keys

The Professional keyboard contains three types of function key: reserved, prelabeled, and generic.

The reserved function keys, F1 and F2 (also known as HOLD SCREEN and PRINT SCREEN), are not available to applications. The other function keys are accessible to applications. All of the function key codes and labels are supplied in the *Tool Kit Reference Manual*.

3.7.6.1 Using Function Keys - The basic Professional keyboard comes with 12 prelabeled function keys and 16 generic function keys (F3 to 14, and F17 to F20). The prelabeled keys are:

ARROWs (4)
DO
FIND
HELP
INSERT HERE
NEXT SCREEN
PREV SCREEN
REMOVE
SELECT

P/OS provides a keyboard label strip that assigns semantic meanings to 11 of the generic keys:

ADDTNL OPTIONS
BREAK
BS
CANCEL
ESC
EXIT
INTERRUPT
LF
MAIN SCREEN
RESUME
SET-UP

If your application provides its own keyboard label strip, you can assign your own semantic meanings to the keys. Otherwise, use the meanings defined by the P/OS label strip, as described below.

- Use ADDTNL OPTIONS to accept auxiliary commands, and to offer services beyond those offered in the current context.

- Use the ARROW keys to move the cursor (or pointer) around the screen in order to select or place objects. For example, a spreadsheet could use the ARROW keys to move to an adjacent cell. In a menu tree, UP ARROW and DOWN ARROW are used in option selection.

- Use BREAK only in commmunications applications to transmit a break character.

- Use DO and CANCEL to confirm and reject input, respectively. Use DO to indicate that a choice has been made or a value entered and that the user is ready to proceed. Use CANCEL to indicate that the choice or value is incorrect and must be reentered. In a menu tree, these keys are used in option selection.

- Use EXIT in a menu tree to back up one menu. If at the top level, use EXIT to terminate the application.

- Use the INTERRUPT/DO sequence to request immediate termination of your application. If the end user presses the INTERRUPT key followed by the DO key, the system aborts all tasks of the running application and, after a specific period of time, removes all application tasks and commons.

  You can process INTERRUPT/DO yourself, or you can allow P/OS to process it for you.

NOTE

> INTERRUPT/DO is equivalent to CTRL/C. To prevent application termination after the INTERRUPT/DO sequence, the task can field the sequence by attaching the terminal for CTRL/C ASTs. See the high-level language documentation for information about how to attach the terminal for CTRL/C ASTs. In MACRO-11, read-pass-all or attaching to the terminal for CTRL/C ASTs can be used to trap CTRL/C. See the *P/OS System Reference Manual* for more information.

- Use MAIN SCREEN to exit your application.

- Use NEXT SCREEN and PREV SCREEN to move through a series of displays. For example, PROSE uses them to move through a document. In a help structure, they activate frame pointers.

- Use RESUME to indicate that the user is ready to continue an interrupted activity. In a help structure, RESUME returns control to the menu or task that activated the structure.

- Use SELECT to record the current position of the cursor (or pointer) for some subsequent action. In a multiple-choice menu, SELECT is used in option selection.

3.7.6.2 Using Function Keys - When a function key (other than DO) terminates a menu, control returns to your application with the following values in the status block:

first word	second word
-14	key code

POSRES also provides several routines to help you process function keys independently from the menu interface. They are:

- **Get Keystroke (GETKEY)**

  This routine inputs a single keystroke from the terminal without echo. The first word of the status block contains one of the following values:

  +1 Indicates that the user pressed a main keyboard key. The second word contains the DEC Multinational decimal code of the key.

  +2 Indicates that the user pressed a function key. The second word contains one of the function key codes.

  -n Indicates that an error has occurred. The second word contains one of the error codes.

- **Parse String (PRSCSI)**

  This routine parses a string for a control sequence (CSI). It scans the string from the left for a CSI character, returns its position in the string, and translates the subsequent characters into a code representing one of the function keys. (See the *Tool Kit Reference Manual*.)

- **Wait for Resume Key (WTRES)**

  This routine echoes all keystrokes except the RESUME key by ringing the keyboard bell. When the user presses the RESUME key, control returns to your application. You can use this routine to allow the user to read something on the screen or change a diskette, for example, before proceeding. Before calling WTRES, display a message such as "Press RESUME to continue" on the screen.

3.7.7 POSRES Task Image Requirements

To use POSRES, you must edit your PAB command file. To determine which edits are required, make a list of the POSRES routines used by your task. Subsequent sections will describe how to edit the .CMD file.

NOTE

Be sure to include any POSRES routines that are used by your high-level language run-time support library. (See your language documentation and Chapter 4.)

Figure 3-9 shows a sample .CMD file, written for a PASCAL program that uses all of the POSRES routines. The contents of this file are explained in detail in the following sections. It is assumed that you are familiar with the general contents and purpose of a .CMD file. If not, please turn to Chapter 4.

```
PASDEM/CP/FP,PASDEM/MA/-SP=PASDEM/MP
CLSTR=PASRES,POSRES,RMSRES:RO
STACK  = 30              ; Start-up stack size
UNITS  = 46              ; Number of units available
GBLDEF = TT$EFN:7        ; Terminal I/O event flag number
GBLDEF = MS$LUN:41       ; Message frame file
GBLDEF = MN$LUN:42       ; Menu Frame file
GBLDEF = MB$LUN:43       ; Message/Status display
GBLDEF = HL$LUN:44       ; Help frame file
ASG    = SY:37           ; P/OS current device
GBLDEF = WC$LUN:45       ; OLDFIL/NEWFIL device
ASG    = TI:38           ; User terminal
GBLDEF = TT$LUN:46       ; Terminal I/O
EXTSCT = MN$BUF:4540     ; Static single-choice buffer
EXTSCT = DM$BUF:4540     ; Dynamic single-choice buffer
EXTSCT = MM$BUF:1000     ; Multiple-choice buffer
EXTSCT = HL$BUF:3410     ; Help frame buffer
EXTSCT = FL$BUF:4310     ; OLDFIL/NEWFIL buffer
//
```

Figure 3-9: PAB Command File with POSRES Options

3.7.7.1 The UNITS Option - The UNITS option specifies in decimal how many logical units (LUNs) your application requires. LUNs refer to simultaneously open files or devices.

3.7.7.2 The GBLDEF Option - The global symbol definition (GBLDEF) option equates a symbolic name to an octal number. POSRES requires a GBLDEF option for each symbol shown in Table 3-1.

Compare the list of POSRES routines used by your task to Table 3-1. Equate a logical unit number (LUN) or an event flag number (EFN) to each symbol associated with each POSRES routine on your list.

For example, if you want to use LUNs 33 through 36 (decimal) for the symbols MS$LUN, MN$LUN, MB$LUN, and HL$LUN, and these LUNs are not used elsewhere, insert the following in your PAB command file:

```
GBLDEF = MS$LUN:41
GBLDEF = MN$LUN:42
GBLDEF = MB$LUN:43
GBLDEF = HL$LUN:44
```

Most POSRES routines require an event flag (EFN) to perform terminal I/O. Select an event flag number in the range 1 to 24 decimal, convert it to octal, and use the GBLDEF option to equate the symbol TT$EFN to the octal number, as follows:

```
GBLDEF = TT$EFN:7
```

This assignment must not conflict with any other event flag assignments (language run-time systems use DIGITAL-reserved event flags 25-32).

3.7.7.3 The ASG Option - The ASG option associates a physical device with one or more logical unit numbers (LUNs). These assignments tell POSRES which devices to use.

The ASG option accepts decimal numbers, but the GBLDEF option accepts octal numbers. To avoid confusion, use this procedure to assign LUNs:

1. Use the ASG option to associate a device with a decimal LUN.

2. Convert the LUN from decimal to octal.

3. Use the GBLDEF option to equate a symbol to the octal LUN.

Available LUNs are in the range from 1 to 128.

NOTE

Your assignments must not conflict with any other ASG assignments in your .CMD file.

Table 3-1: POSRES Global Symbols

Routine	HL$LUN	MB$LUN	MN$LUN	MS$LUN	TT$EFN	TT$LUN	WC$LUN
DMENU	X	-	-	-	X	X	-
FATLER	-	-	-	-	X	X	-
GETKEY	-	-	-	-	X	X	-
HCLOSE	X	-	-	-	-	-	-
HELP	X	-	-	-	X	X	-
HFILE	X	-	-	-	-	-	-
HFRAME	X	-	-	-	-	-	-
MCLOSE	-	-	X	-	-	-	-
MENU	X	-	X	-	X	X	-
MFILE	-	-	X	-	-	-	-
MFRAME	-	-	X	-	-	-	-
MMENU	X	-	-	-	X	X	-
MSGBRD	-	X	-	-	-	-	-
NEWFIL	-	-	X	-	X	X	X
OLDFIL	-	-	X	-	X	X	X
PRSCSI	-	-	-	-	-	-	-
RDMSG	-	-	-	X	-	-	-
WTRES	-	-	-	-	X	X	-

X = symbol used
- = symbol not used

POSRES requires that, in your .CMD file, you assign a LUN to TI:, the Professional keyboard/video monitor. Using the ASG option, equate TI: to a LUN number, and then use the GBLDEF option to equate the LUN number with TT$LUN. An example using LUN number 38 (decimal) follows:

```
ASG = TI:38
GBLDEF = TT$LUN:46
```

Additionally, if you are using the POSRES routines OLDFIL or NEWFIL, you must assign a LUN to SY:, the P/OS current device, and equate it to WC$LUN. For example, if you want to use LUN 37 (decimal) and it is not used elsewhere, insert the following:

```
ASG = SY:37
GBLDEF = WC$LUN:45
```

3.7.7.4 The EXTSCT Option - POSRES uses the program section names shown in Table 3-2 as buffers to store menus, help frames, and so forth. Insert an EXTSCT option for each buffer shown, to extend its program section by a specified (octal) number of bytes.

Compare your list of used POSRES routines to Table 3-2 and assign each accessed buffer sufficient size, as described below. If a routine does not appear in the table, it does not access any buffer. If your task does not use any of the routines that access a particular buffer, assign that buffer a size of zero. Do not omit any buffer names.

Table 3-2: Buffers Accessed by POSRES Routines

Routine	DM$BUF	FL$BUF	HL$BUF	MM$BUF	MN$BUF
DMENU	X	-	X	-	-
DPACK	X	-	-	-	-
HCLOSE	-	-	X	-	-
HELP	-	-	X	-	-
HFILE	-	-	X	-	-
HFRAME	-	-	X	-	-
MCLOSE	-	-	-	-	X
MENU	-	-	X	-	X
MFILE	-	-	-	-	X
MFRAME	-	-	-	-	X
MMENU	-	-	X	X	-
MPACK	-	-	-	X	-
MUNPK	-	-	-	-	X
NEWFIL	-	-	X	-	X
OLDFIL	-	X	X	X	X

X = buffer accessed
- = buffer not accessed

Figure 3-10 shows an example of the EXTSCT options with values calculated for the largest possible frame of each type. In other words, if every field on a menu, help, and message frame were filled in with the suggested maximum amount of data, the respective buffers would have to be allocated the sizes shown.

```
EXTSCT = DM$BUF:4540    ; Dynamic single-choice buffer
EXTSCT = FL$BUF:4310    ; OLDFIL/NEWFIL buffer
EXTSCT = HL$BUF:3410    ; Help frame buffer
EXTSCT = MM$BUF:1000    ; Multiple-choice buffer
EXTSCT = MN$BUF:4540    ; Static single-choice buffer
```

Figure 3-10: Suggested Maximum POSRES Buffer Sizes

NOTE

There is no maximum size for FL$BUF or MM$BUF. If you use extraordinarily large menus, you might have to expand these buffers.

You should begin with the maximum size for each accessed buffer. If your task exceeds the bounds of virtual memory, you can reduce the size of one or more buffers as described below.

Some care should be exercised when shrinking POSRES buffers. If you change a frame file, you might find that a frame has become too large for its buffer, requiring you to enlarge the buffer and task build again. For example, translating frames into another language can cause them to expand. Therefore, allocate some extra space to any buffer for which the maximum frame size is not completely stable.

To compute the minimum buffer for a static display (a static single-choice menu, help menu, and so forth), do the following:

1. Take the size of the largest frame (provided by FDT in decimal when you use the CONVERT command). If the source data file is unavailable, you can use a system utility to analyze the frame file. On a VAX/VMS system, type the following:

   ```
   $ ANALYZE/RMS file.typ
   ```

   The longest record in the file appears in the "RMS FILE ATTRIBUTES" section of the output. On either RSX-11M/M-PLUS or the PRO/Tool Kit, type the following:

```
$ RUN $DMP
DMP> TT:=filename.typ/HD/BL:0
```

The length of the longest record will appear in the F.RSIZ field in the "HEADER AREA" of the output.

2. Add 200 bytes to this number.

3. Convert the result to octal.

4. Use this number with the EXTSCT option.

You can approximate the minimum buffer size for a dynamic single-choice menu by using FDT to build a similar static single-choice menu and using its converted size. Choose a frame that represents your worst (largest) case situation and allow a large margin for error (approximately 200 bytes). Otherwise, use this procedure:

1. Total the sizes of all the fields in the largest frame.

2. Add an overhead of eight bytes per field.

3. Add 200 bytes to the total.

4. Convert the result to octal.

5. Use this number with the EXTSCT option.

The minimum buffer size for a multiple-choice menu is difficult to determine because it depends on factors present at run-time. If you find it necessary to reduce this buffer, the recommended procedure is to change the buffer size by the desired amount, task build, and test your application thoroughly using the largest possible multiple-choice menu.

NOTE

The following section assumes that you are familiar with Overlay Descriptor Language. If not, it is recommended that you refer to the *RSX-11M/M-PLUS and Micro/RSX Task Builder Manual* before continuing.

3.7.7.5 Placing Buffers in Overlay Branches - Another way to reclaim virtual memory is to place all code that refers to POSRES routines and buffers into overlay segments. Unless you specify otherwise, PAB allocates POSRES buffers in the root.

If you put a POSRES buffer in an overlay segment:

- The buffer's contents will be reinitialized whenever that segment is loaded into memory. Thus, you are responsible for replacing its contents.

- You cannot call the POSRES routine that uses that buffer from another overlay segment (except from another co-tree). Thus, you must subdivide your task into segments that use POSRES and those that do not.

Figure 3-11 shows an example of how to place the buffers into an overlay branch.

```
          .ROOT RCODE-RMSROT-(*MENU,*CODE)
MENU:     .FCTR MCODE-BUFFA-BUFFB
CODE:     .FCTR ARTN-BRTN-CRTN
@LB:[1,5]RMSRLX

          .PSECT DM$BUF,RW,D,GBL,REL,CON
          .PSECT DM$BUG,RW,D,GBL,REL,CON
          .PSECT FL$BUF,RW,D,GBL,REL,CON
          .PSECT FL$BUG,RW,D,GBL,REL,CON
          .PSECT FL$FAB,RW,D,GBL,REL,CON
          .PSECT HL$BUF,RW,D,GBL,REL,CON
          .PSECT HL$BUG,RW,D,GBL,REL,CON
          .PSECT MM$BUF,RW,D,GBL,REL,CON
          .PSECT MM$BUG,RW,D,GBL,REL,CON
          .PSECT MN$BUF,RW,D,GBL,REL,CON
          .PSECT MN$BUG,RW,D,GBL,REL,CON

BUFFA:    .FCTR DM$BUF-DM$BUG-FL$BUF-FL$BUG-FL$FAB-HL$BUF
BUFFB:    .FCTR HL$BUG-BM$BUF-MM$BUG-MN$BUF-MN$BUG
          .END
```

Figure 3-11: Sample .ODL File Showing Overlaid Buffers

In Figure 3-11, the modules RCODE, MCODE, ARTN, BRTN and CRTN all refer to user-supplied portions of the task. Use the .PSECT directives exactly as shown. (If they do not match the actual attributes of the PSECTs that they refer to, you will get errors when task building.)

The POSRES buffers that you can overlay are shown below:

Buffer	Use
DM$BUF, DM$BUG	Dynamic menu
FL$BUF, FL$BUG	OLDFIL, NEWFIL
FL$FAB	OLDFIL, NEWFIL
HL$BUF, HL$BUG	Help frames
MM$BUF, MM$BUG	Multiple-choice menu
MN$BUF, MN$BUG	Static, single choice menu

The xxBUF/xxBUG pairs must be in the same overlay. These pairs of PSECTS are used by POSRES to determine buffer sizes. If they are not in the same overlay, unpredictable behavior will result.

CHAPTER 4

BUILDING THE APPLICATION TASKS

The Professional Application Builder (PAB) is a utility that links your object modules (.OBJ files) with system software, producing task image files that can be executed on P/OS.

This chapter provides enough practical information about PAB to task build simple applications. Your documentation set includes the *RSX-11M/M-PLUS and Micro/RSX Task Builder Manual*, which describes PAB in detail.

If you are developing large or sophisticated applications, you should read the manual and become proficient in the PAB languages. PAB has features that allow you to optimize task images in many different ways.

4.1 INVOKING PAB ON THE PRO/TOOL KIT

If you are using the PRO/Tool Kit, invoke PAB with the LINK command, which is described in the *PRO/Tool Kit Command Language and Utilities Manual*.

A sample LINK command in DCL follows:

```
$ LINK @file
```

The default file type is .CMD. Alternatively, you can let the system prompt you for the filename:

```
$ LINK
File(s)? @file
```

4.2 INVOKING PAB ON VAX/VMS

On VAX/VMS, the PAB executable image is called PROTKB.EXE. It

runs under VAX-11 RSX, and it supports named as well as numbered directories. Insert the following symbol in your LOGIN.COM file:

```
$ PAB :== $PROTKB
```

Once the symbol is defined, you can use a single command to invoke PAB:

```
$ PAB @file
```

The default file type is .CMD. Alternatively, you can type:

```
$ PAB
PAB> @file
```

4.3 INVOKING PAB ON RSX-11M/M-PLUS (DCL)

On RSX-11M/M-PLUS, the PAB executable image is called PROTKB.TSK. If PAB is installed on your system as "...PAB", you can invoke it with the following command:

```
$ PAB @file
```

If PAB is not an installed task on your system, invoke it with the following command:

```
$ RUN $PROTKB
PAB> @file
```

The same commands also work for MCR.

4.4 BUILDING APPLICATIONS

You must create two PAB files--a .CMD file and an .ODL file--for each task image in your application. Each file contains commands in a different language that tell PAB exactly how to build the desired task image.

Some of the Tool Kit languages include software for creating PAB files that automatically contain language-specific information as well as information for PRO/RMS-11. Others simply describe the required files and expect you to create them yourself with an editor. In either case, create the required files and examine their contents and format.

Because no language can anticipate all of your task's requirements, you will almost certainly have to edit those files before task building. The following services require specific information in your PAB files:

- CORE Graphics Library
- PRO/FMS-11
- PRO/RMS-11
- POSRES User Interface Library
- POSSUM System Services

If your task uses any of those services, refer to the documentation for each and make the appropriate edits.

NOTE

Most of the high-level languages use POSRES User Interface Library routines (particularly RDMSG and WTRES) for run-time support and thus require some POSRES support in your .CMD file. POSRES is described in detail in Chapter 3.

4.5 THE COMMAND (.CMD) FILE

The PAB command (.CMD) file specifies input and output files and contains option lines that specify cluster libraries, buffer sizes, logical unit number assignments, and so forth.

Figure 4-1 shows a sample .CMD file, written for a MACRO program, that uses POSSUM and PRO/RMS-11.

```
SAMPLE=SAMPLE/MP
CLSTR=POSSUM,RMSRES:RO
//
```

Figure 4-1: Sample PAB Command File

The double slash (//) indicates the end of the file. The rest of the file is explained in the following sections. Refer to Figure 4-1 in each of the next sections.

4.5.1 The Command Line

The first line in a command file is a command that specifies input and output files and switches. Input files are on the right side of the equal sign and output files are on the left:

```
SAMPLE=SAMPLE/MP
```

This command specifies one input file and one output file, both named SAMPLE. The /MP switch specifies that the input file is an overlay descriptor language (.ODL) file, as described in Section 4.6. The output file is a task image (default file type .TSK).

The use of an .ODL file does not imply that your task must be overlaid to run on P/OS. It simply describes your task image in more detail than is possible in the .CMD file.

If you would like a map that shows exactly how your task loads into memory, specify a second output file, such as:

```
SAMPLE,SAMPLE/MA/-SP=SAMPLE/MP
```

This command specifies a task image and a load map (.MAP) file. The /MA switch specifies that the load map contain the names of the system library routines your task uses. The /-SP switch prohibits automatic printing of the load map on Host Tool Kit systems.

4.5.2 The CLSTR Option

The CLSTR option specifies the clustered resident libraries used by your task. If your task refers to a non-null-rooted clustered resident library, it must be the first (default) library in the CLSTR option. This applies to high-level language run-time libraries, such as PASRES and PBFSML. If all are null-rooted, the first library called by your task becomes the default.

For example, if a PASCAL program uses PRO/RMS-11 and POSRES services, the CLSTR option would be:

```
CLSTR=PASRES,POSRES,RMSRES:RO
```

NOTE

Do not embed any spaces in the CLSTR option. Also, do not include a comment on the CLSTR option. It causes PAB to return a fatal option syntax error.

The :RO switch specifies read-only access. The overlay run-time system allows read-write access to nondefault (in addition to default) clustered libraries that have not been installed read-only. Such libraries can be useful for:

- Passing information between cooperating application tasks. (The tasks should provide their own access synchronization.)

- Extending the effective available read-write virtual memory usable for a task's impure data.

In both cases, the tasks must ensure that the library is mapped by calling a routine in the library that does not return to the caller until access to the read-write data is completed. This is not normally necessary for a non-null-rooted default cluster member, since it is usually already mapped as desired.

4.5.3 NULLIB

The special non-null-rooted default cluster member NULLIB is provided for two purposes:

- It can guard against potential memory fragmentation problems that can cause task deadlock.

- It can provide better performance in cases when a null-rooted cluster member would otherwise become the effective default member of the cluster and would be unnecessarily remapped (and potentially reloaded from disk). Assuming that the application would access cluster members other than the first one accessed, remapping that first member after every access to some other member could prove costly.

As a general rule, when all members of a library cluster have null roots, your application should attempt to ensure that the first library accessed (which will become the effective default cluster member) is the one that the application will refer to most frequently. This will minimize the likelihood of unnecessary mapping.

4.6 THE OVERLAY DESCRIPTOR LANGUAGE FILE

The overlay descriptor language (.ODL) file specifies the object modules used by your task (some of which are automatically extracted from libraries) and describes how your task image will use its 64KB of virtual memory.

Each high-level language and development tool specifies which object modules and libraries to include in your .ODL file, along with your own object modules.

NOTE

> Some tools might say to use files in LB:[1,1], the RSX system library directory. The Host Tool Kit system library directory is LB:[1,5], not LB:[1,1]. Thus, PAB automatically replaces all references (including defaults) to LB:[1,1] with LB:[1,5].

Normally, PAB allocates memory for each object module in linear fashion (while automatically extracting object modules from SYSLIB.OLB as needed). In order to reduce the size of a task, you can create a structure where one or more modules are overlaid (share the same memory). That subject is discussed in detail in the *RSX-11M/M-PLUS and Micro/RSX Task Builder Manual*.

The following is an example of an .ODL file, written for a MACRO-11 program that uses PRO/RMS-11:

```
        .ROOT   USER-RMSROT
USER:   .FCTR   FIRST-USERSUB
@LB:[1,5]RMSRLX
        .END
```

The first line is a .ROOT directive, which defines the structure and contents of the task image. Although it can contain object modules, .ROOT usually contains references to symbols defined elsewhere in the .ODL file, as is the case here: USER and RMSROT are symbols.

The second line is a .FCTR directive, which defines the symbol USER as two object modules: FIRST and USERSUB. To specify additional object modules, you could add them to this .FCTR directive or include additional .FCTR directives as needed.

The hyphen specifies that FIRST and USERSUB are to be concatenated (each is to have its own area of memory). A comma would specify that the modules are to be overlaid. In that case, PAB would allocate a single area of memory, usable by both modules, but only one at a time.

The third line includes the PRO/RMS-11 overlay descriptor language file RMSRLX.ODL, which defines the symbol RMSROT. The at-sign character (@) allows you to nest .ODL files in the same way that you can nest program source code.

CHAPTER 5

TESTING THE APPLICATION

Follow these steps to test your application:

1. **Create application directory**

   Create a directory to contain the executable files and installation command file. See Section 3.1 for details.

2. **Transfer files to target system (Host Tool Kit Only)**

   Omit this step if you develop your application using the PRO/Tool Kit.

   To transfer files from the host system, invoke the Terminal Emulator, use the Professional File Transfer (PFT) utility to copy your application files to the application directory on your Professional, and exit from the Terminal Emulator. (See the *PRO/Communications Manual* for details.)

NOTE

For the second and subsequent test runs, you need transfer only files that have changed since the last test. P/OS uses the latest versions of files by default if you do not supply a version number for the file.

3. **Install the application**

   Use the Fast Install utility described in the *Tool Kit Reference Manual*.

NOTE

Once you have installed your application, it is unnecessary to reinstall it when you copy new files to your application directory.

4. **Run the application**

   Select the application from the menu on which you installed it. If it won't start up or aborts, check the P/OS error codes listed in the *Tool Kit Reference Manual*. Also, make sure that the installation command file installs all necessary tasks.

5.1 DEBUGGING WITH A SECOND TERMINAL

Some high-level languages provide interactive debugging software that allows you to control program execution, manipulate variables, and so forth. Refer to your language documentation for more information.

Included with the Tool Kit is ODT, a debugger that you can use with any progamming language, although it is intended mainly for use with MACRO-11 programs.

The Tool Kit allows you to connect a separate terminal to your Professional in order to redirect ODT output. You can redirect debugger I/O to the debugging terminal, while your application has undisturbed use of the Professional screen and keyboard. This is especially useful when debugging programs that use the Professional video screen for menu interfaces, graphics, and so forth.

To debug with a second terminal, do the following:

1. Use a BCC08 console cable or BCC05 printer cable to attach your choice of debugging terminal to the printer port on the back of the Professional system unit.

   NOTE

   The console cable is for debugging only. If you want to use an LA50, LQP02, or LA100 printer for normal printing purposes, use a printer cable.

2. Set the second terminal to run at 9600 baud if using a console cable; for the printer cable, use 4800 baud.

3. Enter the following DCL command from the first terminal:

```
$ ASSIGN TT002: CL000:
```

5.2 DEBUGGING THE DISTRIBUTION KIT

When your application is ready to distribute, you should create a distribution kit using the Application Diskette Builder (described in the *Tool Kit Reference Manual*). This places all the files of the application onto one or more diskettes.

You should then debug the distribution kit itself. Remove all traces of the installed application from your Professional and then reinstall the application from your distribution kit. Test the application to ensure that all the necessary files have been installed from the distribution diskette.

Always completely remove the application files and the directory in which you developed the application (if developed on PRO/Tool Kit) before testing the distribution kit. There can be references to the development directory name hidden in your code. You might have placed files in system directories and failed to include these files in the distribution kit.

If you do not remove these files and the development directory before installing the application from the distribution kit, your testing can appear to be successful, yet when the kit is installed on a user's system, it will fail.

You might want to test the distribution kit on another Professional to guarantee that your application does not depend on a file or directory that exists only on your development system.

CHAPTER 6

TUNING THE APPLICATION

Application tuning is the process of increasing the efficiency of your code, by reducing unnecessary operations or by making better use of system features. This chapter provides ideas that you can use to help tune your application.

6.1 EXTTSK OPTION VERSUS EXTK$ DIRECTIVE

The EXTK$ directive requires a great deal of overhead when the system checkpoints task regions to disk and reallocates memory in the system controlled GEN partition.

Use the PAB option EXTTSK in your task build command file instead of using the EXTK$ directive in your task whenever possible.

If you must use the EXTK$ directive, you should extend your task in large increments to reduce the overhead in the checkpointing activity.

6.2 DIRECTIVES VERSUS SERVERS

There is a small area of overlap between callable system routines and several system directives. The logical name support directives and the default directory directives can both be manipulated, by a callable system routine or directly by system directives. If you are programming in a language that allows you to issue system directives, then you should consider issuing the directives rather then calling the system routine.

Let's examine the specific cases. The system routine PROLOG performs logical name operations; to perform the operations, the routine translates the logical name strings to uppercase and issues one of the system directive CLOG$, DLOG$, or TLOG$ to perform the operation.

However, your task could simply issue the directive itself and save time.

If your task issues the directive itself, it must specify LT.USR and a modifier of 0 and ensure that the logical is translated to uppercase if used by RMS-11 or another task or system service. Logical strings are binary and hence case-sensitive. That is why the server always translates to uppercase first.

If you use the callable routine, the task that services the request must be loaded into memory and started; it must receive and process the request using a system directive; and then it must return the results to the calling task. For logical name support, the additional services provided by the callable routine ensure that the string is in uppercase and specify LT.USR and modifier 0.

Default directory operations, on the other hand, should be accomplished using the PROLOG routine in POSSUM. PROLOG performs RMS parsing operations on the input string to determine that the string contains a valid device and directory specification. The RMS parser verifies that the device syntax is legitimate.

If your task issues the directives itself, there is a possibility that it could specify an invalid directory string--the directive accepts any string as the default directory string. This could cause problems when attempting to find or create files.

6.3 FILE HANDLING

Several file-handling techniques can increase application performance. Almost all of these methods involve some trade-offs, either in terms of some additional logic in the task or in the use of disk space.

6.3.1 When to Open Files

One of the simplest techniques to use when handling files is to defer opening a file until necessary. It might seem simpler to open all files at application start-up, but the primary tuning goal at application start-up is to begin interacting with the user as soon as possible. Opening files takes time, and if possible should be done when data is actually needed from the file.

6.3.2 Use of File IDs

The use of file IDs for operations on files reduces overhead. If possible, preserve the file ID of a file that is opened several times. This incurs less overhead than finding the file by name each time. The cost to your application is merely to remember the file ID. It would be even more time efficient to simply keep the file open.

6.3.3 File Preallocation

Preallocating sufficient media space for a file saves time and cuts overhead. If a file does not contain sufficient room to append records, then the system must allocate the necessary disk space to the file. By preallocating the amount of space that the file will require when the file is created, your task spares the overhead of extending the file later.

The technique of preallocating sufficient media space only works when you can forecast the size of the file to be written. However, if you know even the minimum amount of space the file will require, then preallocating that minimum amount saves time later.

6.3.4 Preextending

An additional method that improves file performance is to specify a reasonable default extend quantity to RMS. This means that whenever RMS needs to extend your file to add an additional record, it extends the file by the number of blocks specified in the default extend quantity.

If the extend quantity is reasonably large, fewer extensions to your file occur. This results in improved performance for your application.

Note that RMS truncates the file to its logical EOF if the last extend is implicit. That is, if no extensions occurred, or if the last extend was caused by the task asking RMS for an explicit extension, then the file will not be automatically truncated on close.

6.3.5 Multiblock I/O

As with most I/O operations, the more data transferred per operation, the more efficient the operation becomes. This rule applies to disk I/O, whether it consists of direct reads and writes to the disk, or of reads and writes through an intermediate record processor such as RMS.

Increasing the size of the reads and writes always reduces the time per block spent reading or writing buffers to the disk. Note, however, that this technique can increase the size of your task's virtual address space.

6.4 VIDEO PERFORMANCE

Video performance is one of the most sensitive areas for application tuning. Regardless of an application's timed speed, if the video throughput appears slow, then that application will be thought of as slow.

There are a number of simple rules and techniques that help tune for the best possible throughput.

6.4.1 Size of Buffer

The size of the output buffers is one of the more important factors governing throughput rate. Very simply, the larger the buffers, the less time spent in overhead and the greater the thoughput rate. Above 64 characters per buffer the output rate approaches the maximum throughput rate of the video.

6.4.2 Buffering

You can achieve significant improvements in terminal output by using large buffers. Intermediate buffering is one technique that you can use to increase the size of the output buffers for the terminal.

Intermediate buffering uses a routine that accepts text for output to the screen. The routine places this text into an intermediate buffer. When the intermediate buffer is full, its contents are written to the terminal.

You also provide a routine to force output to the terminal, thus allowing the forced flushing of the intermediate buffer.

Let's take a look at a specific example of intermediate buffering. Assume a task generates output of varying lengths and at varying times. Not all output strings generated by the task are terminated by a carriage-return and line-feed (a prompting string, for example).

Figure 6-1 shows a listing of sample routines that handle intermediate buffering. The code shown in Figure 6-1 holds all text in the buffer until the buffer is full, or until the routine to force output is called.

Circumstances under which the force output routine might be called are:

- At the end of any escape sequence, to prevent it from being broken into two separate QIOs.

- In the OUTCHR routine, whenever a carriage-return or line-feed character is detected. This might seem to violate the rule of buffering, but forcing out buffers on line terminator characters can make the output appear smoother to the user.

- In an AST routine specified in a MRKT$ directive, which is called a few times a second to dump the buffer. If text data appears at random intervals, and without line terminators to rely on, this can be the only way to output a sequence that does not end with a line terminator (such as a prompting sequence).

  The rate at which the you should call the AST routine depends on the rate at which data appears, as well as the amount of overhead caused by the AST routine.

6.4.3 Turning the Cursor Off

Another way to save system overhead and increase the amount of CPU time available to your task is to turn the cursor off. You turn off the cursor by using the appropriate escape sequence. As a result, the terminal task no longer needs to blink the cursor.

```
; OUTCHR - This routine sends a character to the terminal. All
;          output is temporarily stored in an intermediate
;          buffer before queuing to the terminal.  The routine
;          assumes explicit carriage control in text strings.
;
;          Input:   R0 - Character

OUTCHR: DEC     TTYCTR          ; Decrement the count
        BLT     OUTCH0          ; No room left, make some
        MOVB    R0,@TTYPTR      ; Store the byte
        INC     TTYPTR          ; Increment pointer
        RETURN                  ; Then return to caller
OUTCH0: QIOW$C  IO.WVB,TT$LUN,TT$EFN,,,,<TTYBUF,.LNTTY,0>
        MOV     #.LNTTY,TTYCTR  ; Reset the counter
        MOV     #TTYBUF,TTYPTR  ; and the pointer
        BR      OUTCHR          ; Try again to output char

; STRING - Prints an ASCIZ string on the terminal.
;          Input:  R0 - Pointer to string

STRING: CALL    $SAVAL          ; Save registers
        MOV     R0,R5           ; Save up pointer
STRIN0: MOVB    (R5)+,R0        ; Get next byte
        BEQ     STRIN1          ; Done
        CALL    OUTCHR          ; Output the character
        BR      STRIN0          ; Loop
STRIN1: RETURN                  ; Return to caller

; FRCOUT - This routine forces out the TTY buffer

FRCOUT: MOV     #.LNTTY,R0      ; Get buffer length
        SUB     TTYCTR,R0       ; Compute amount filled
        BEQ     FRCOU0          ; None, just exit
        QIOW$S  #IO.WVB,#TT$LUN,#TT$EFN,,,,<#TTYBUF,R0,#0>
        MOV     #TTYBUF,TTYPTR  ; Reset pointer
        MOV     #.LNTTY,TTYCTR  ; and counter
FRCOU0: RETURN                  ; Return to caller

; Terminal buffer
        .EVEN
TTYCTR: .BLKW   1               ; Buffer byte count
TTYPTR: .BLKW   1               ; Byte pointer
TTYBUF: .BLKB   80.             ; TTY buffer
 .LNTTY = .-TTYBUF              ; Length of buffer
```

Figure 6-1: Intermediate Buffering

Note that you should not turn the cursor off in every possible instance. Consider how the screen appears without a cursor. For some applications, such as a spreadsheet, the absence of a blinking cursor might not be a problem. For other applications, turning off the cursor can startle the user and possibly even make it appear that system is not operating.

6.4.4 Common Video Techniques

There are several fairly common techniques that can be used effectively when dealing with video output. The savings in performance and overhead can be large.

- **Write only necessary data**

  The first technique is simply a matter of not writing more data than is necessary. For instance, your task can manage the entire screen and provide updates only to certain portions of the screen. An inefficient method of updating the screen is to rewrite the entire screen on every update. That is rather slow, since the amount of data could be large.

  A variation is to only update any regions that are known to change dynamically. That is faster, but could still produce a large amount of output.

  What is needed are two copies of the dynamic sections of the screen image in your task. One copy reflects the current state of the screen, and the other copy reflects the desired state. Your task compares these memory-resident copies and determines which character positions must change and only updates those positions.

  This technique reduces the amount of I/O that must be performed to an absolute minimum, which in turn will increase application performance and terminal throughput.

  The cost of this technique is in the virtual memory space required to contain two copies of any dynamic regions on the screen. Two copies of an 80-column, 24-line screen require 3840 bytes. For some tasks, this can be too much. For others, the performance improvement is well worth the added task size.

- **Use cursor motion escape sequences**

  Use cursor motion escape sequences when the cursor position is changing rapidly within a row or column. Here the cursor motion escape sequences (such as UP, DOWN, LEFT, RIGHT) are more efficient than the cursor positioning sequence.

  Using the cursor motion sequences works well in conjunction with the multiple-screen technique. Combined, they can result in tremendous savings in the amount of data that must be output, and a corresponding increase in terminal throughput.

  A corollary method is to take advantage of the editing functions provided as part of the VT200 interface of the video. These include line insert, line delete, character insert, character delete.

- **Clear screen and set cursor to home**

  The third technique also applies only to applications that manage the entire screen as well as producing large amounts of output.

  It is known that output that causes the screen to scroll appears on the screen at a slower rate than output that does not cause the screen to scroll. You can use this to advantage when writing output to the screen. If the application produces logically distinct, single screens of output, then improve performance by clearing the screen and homing the cursor before writing to the screen.

  This technique has no penalty in the form of increased virtual address space, but you should consider how your application appears when screens of output are presented without scrolling.

6.5 MEMORY MANAGEMENT CONSIDERATIONS

An application can benefit from the use of the memory management, or programmammable logical address space (PLAS) directives. You can use these directives to modify virtual address space or to create or map additional regions that contain data.

The most common problem is misuse (or over-use) of the PLAS directives. A single large data region could cause problems when your program attempts to bring the region in and out of memory.

Another problem can result from sending task regions by reference, using the SREF$ directive. The sender, receiver, and all regions mapped by both, must be in memory. This can cause severe memory contention and perhaps even a deadlock situation.

Another problem to consider is the use of some PLAS directives that can waste system resources. Use WS.NAT in the window block for a RREF$ directive to save system pool space and improve performance.

Also, use the fast remap capability, described in the *P/OS System Reference Manual*, to improve performance.

6.6 MULTITASK APPLICATIONS

Multitask applications have special problems. These difficulties result from intertask communication, and memory or CPU contention.

6.6.1 Significant Event Impact

If two cooperating tasks within an application are using global event flags to synchronize themselves to events, then the proper method of setting a flag for the other task to react to is to declare a significant event after setting the flag.

The following simple representation shows two cooperating tasks setting flags without the significant event mechanism.

```
Task T1      Task T2
----------   ----------
SETF$S #34   WTSE$S #34
WTSE$S #33   CLEF$S #34
CLEF$S #33   (code here)
             SETF$S #33
```

The next simple representation shows two cooperating tasks setting flags using the significant event mechanism.

Note that the only difference is the declaration of a significant event after a flag has been set. This simple change provides significant performance improvement.

Task T1	Task T2
SETF$S #34	WTSE$S #34
DECL$S	CLEF$S #34
WTSE$S #33	(code here)
CLEF$S #33	SETF$S #33
	DECL$S

6.6.2 NULLIB

NULLIB can provide improved performance in cases when a null-rooted cluster member might become the effective default cluster member, and would be unnecessarily remapped. See Section 4.5.3 for details.

6.6.3 Contention

If an application has multiple active tasks, you should avoid both memory and CPU contention problems. If it is not necessary for a task to run concurrently with other tasks, and when one of the application tasks must be available to perform some processing, you should stop the task instead putting it into a wait state.

With directives that initiate a wait state, the task remains in competition for memory resources and could possibly create contention problems. Directives that cause the task to be stopped remove the task from contention for memory and allow easier access for the other tasks that must still perform processing.

In addition, tasks that stop can be checkpointed and remain in the checkpoint file until the stop bit is cleared. They will not compete with other tasks to be brought back into memory. To avoid possible memory fragmentation, the task should not be mapped to any commons while stopped.

Specifically, assume that you have a server task that is notified of an operation by an event flag. Rather than using WTSE$, use STSE$. When the event occurs, your task will become unstopped and will start executing and competing for memory again. If it has been checkpointed, it will be brought back into memory.

Once the task has completed its function it can stop itself once again by using the STSE$ directive.

6.7 POOL CONSIDERATIONS

The system's dynamic storage region, or pool, can be a source of performance problems. You should be aware of functions that can result in pool depletion.

An application can cause the system to exhaust all available free space in pool. The following sections describe circumstances that can lead to pool exhaustion.

6.7.1 Offspring Control Blocks

Offspring control blocks are packets that build up and exhaust pool. When a parent task exits, the OCBs of its offspring are not deallocated. The OCBs are deallocated only when the offspring exits or emits status (EMST$).

If you request a task (using RQST$ or RPOI$ without RP.OAL), rather then spawning it, no OCB is created.

6.7.2 Lock Blocks

If your application uses QIOs rather than RMS for file access, then you should exercise great care.

For example, suppose your application uses RMS to open a file, but uses QIOs to read and write blocks in that file. A possible side effect is that the improper setting of the SHR field in the RMS FAB block can cause many record lock blocks to be generated.

This eventually depletes system pool, causing the system to generate confusing messages. You must use caution when selecting the values to be placed in the SHR field.

If you use RMS for all file accesses, then RMS manages all the record lock blocks and system pool will not be unnecessarily depleted.

6.7.3 Open Files

Any open file requires that the system use some amount of pool space. If you have many files open, you might be using a lot of pool.

To reduce pool requirements, try to reduce the number of files open at any one time. Any files open for write operations could be made unusable if your task aborts.

6.7.4 Attachment Descriptor Blocks

Attachment descriptor blocks can build up in pool, unless you specify that the blocks should be created only when necessary.

For instance, use WS.NAT in the window block for a RREF$ directive. This tells the directive that the attachment descriptor block should be created only when necessary.

6.7.5 Send Data Packets

Packets from the send data directive accumulate in secondary pool if the receiver is not receiving the packets. Make sure the receiver is at least receiving the packets, and if so, that it is receiving them at the same rate at which the packets are being sent.

6.7.6 I/O Packets

It is easy to fill up pool with I/O packets. If you generate a large number of QIO$s to a relatively slow device, such as the line printer port (TT2:), you can quickly fill up pool.

To avoid the problem, make sure you provide code to check for the completion of any QIO$ directive that your program calls.

6.8 BUFFERED INPUT AND ASTS WITH NOTIFICATION

You can optimize terminal input by requesting the terminal driver to return all currently available data, rather than reading input on a character-by-character basis. To request that the terminal driver return all available data, specify an input QIO with a TF.TMO option whose timeout value is 0.

Even if your task processes input on a character-by-character basis, you can still obtain all available characters by establishing the following buffering scheme: one subroutine fills an intermediate buffer from terminal input, and another subroutine empties the buffer as the characters are processed.

The following is a simple example that uses a ring buffer to hold the characters.

```
        .TITLE  SAMPLE PROGRAM
;+
; This sample program demonstrates the use of notification ASTs
; and terminal driver input with timeout=0. It is for
; demonstration only and is not a fully working program.
;-
        .MCALL  EXIT$S
        .MCALL  DIR$, CLEF$, SETF$, WTSE$, DSAR$S, ENAR$S
        .MCALL  QIOW$, ALUN$, ASTX$S

BUFEFN = 1
QIOEFN = 2
TTLUN = 5

START:
        CALL    INIT            ; initialize miscellaneous
        BCS     20$             ; initialization error--
                                ; can't continue
        CALL    PROGRM          ; do any preprocessing
;
; The next routine will read a character.  If there are no
; characters currently available, it will wait for one.
;
5$:
        CALL    GETCHR          ; read character
10$:
        CALL    PROCESS         ; process the character
        BCC     5$              ; if CC, we want more
20$:
        EXIT$S                  ; else exit

;+
; INIT
;
; One-time initialization:
;       1.  There are two pointers into the ring buffer
;               RDHERE - the address of the next available byte
;                        in the buffer for terminal input
;               TAKENX - the address of the last input byte
;                        processed by the task
;               When (RDHERE) = (TAKENX)+1 there is no data for
```

```
;                       task to process.
;       2.  The event flag, BUFEFN, is set if there is any data in
;           the buffer for the task to process.
;       3.  The AST will just be for notification.  The AST routine
;           will ignore the top of the stack parameter.
;       4.  WRAPFL will indicate whether data is coming in faster
;           than it can be processed.
;-
INIT:
        MOV     #RINGBF+1,RDHERE ; first QIO data goes here
        MOV     #RINGBF,TAKENX  ; mark the fact that there
                                ;   is no data to be processed
        CLR     WRAPFL          ; hasn't wrapped yet
;
; The event flag will be set by the AST routine to alert the
; task that there is data in the buffer
;
        DIR$    #CLEF           ; clear buffer data event flag
        BCS     20$             ; can't clear event flag
;
; Attach for input ASTs, notification only.  Do not process
; CTRL-C if doing a read all.
;
        DIR$    #ALUN           ; assign the LUN
        BCS     20$             ; error
        DIR$    #QIOATT         ; attach
        BCS     20$             ; error in directive
        TSTB    IOSB            ; error in I/O?
        BGT     20$             ; no, carry is clear
        SEC                     ; indicate I/O error
20$:
        RETURN

;+
; GETCHR
;
; This routine is entered when the application is
; ready to process a character.  The event flag
; BUFEFN is set by the input routine if there is
; any data in the buffer.  If this event flag is
; not set, the routine will wait until it is set.
;
; Input:        RDHERE contains the address of the next location
;                       into which to read a character
;               TAKENX contains the address of the last
;                       character processed by this routine
;
; Output:       R0 (low byte) contains the character
;               If no characters are left in the buffer,
;               Then    (RDHERE) = (TAKENX)+1
;                       BUFEFN will be cleared
```

```
;-
GETCHR:
;
; The AST routine will set BUFEFN when there are
; characters in buffer.
;
        DIR$    #WAIT           ; wait for buffer to be filled
;
; Now we know we have some input
;
        DSAR$S                  ; disable ASTs until we can
                                ; extract a character
        INC     TAKENX          ; update pointer
        CMP     TAKENX,#RINGEND ; at end of ring buffer?
        BLO     10$             ; if LO, no
        MOV     #RINGBF,TAKENX  ; point to beginning of buffer
10$:
        MOV     TAKENX,R0       ; get address of next character
        MOVB    (R0)+,-(SP)     ; hold character and point to next
        CMP     R0,#RINGEND     ; at end of buffer now?
        BLO     20$             ; if LO, no
        MOV     #RINGBF,R0      ; yes, wrap pointer
;
; At this point R0 -> the next character to
; be processed. If this address is the same
; as the next available byte in the ring
; buffer for input, then we have processed
; all the characters. Signal by clearing
; the event flag.
;
20$:
        CMP     R0,RDHERE       ; all characters out?
        BNE     30$             ; no
        DIR$    #CLEF           ; yes, signal buffer empty
30$:
        MOVB    (SP)+,R0        ; retrieve character
        ENAR$S                  ; allow ASTs again
        RETURN
;+
; ASTADR
;
; Input notification AST handling routine
;
; Read with a timeout of 0, which will return all the
; characters in the typeahead buffer up to the size of our
; buffer.  Move these characters to the ring buffer and set the
; event flag which indicates that there is data in the buffer.
;
; Output:       If read succeeds, ring buffer is filled
;                       and RDHERE points to next available byte.
;               All registers preserved.
```

```
;-
ASTADR:
        MOV     R0,(SP)         ; replace unused AST parameter on
                                ; stack with a working
                                ; register to save
        MOV     R1,-(SP)        ; save another
        MOV     #TMPBUF,R0      ; I/O buffer
        MOV     #IOSB,R1        ; I/O status block

        DIR$    #QIORNE         ; read with timeout of 0
        BCS     30$             ; read didn't work
        TSTB    (R1)            ; success?
        BLE     30$             ; no
        MOV     2(R1),R1        ; get number of characters
        BEQ     30$             ; just in case

        DIR$    #SETF           ; indicate buffer has characters
10$:
;
; At this point we must check to make sure that we
; don't wrap around to data which has been read in
; but not processed.  Because the character at the
; next address after TAKENX has not been processed,
; we cannot read another character if it would mean
; that RDHERE would pass TAKENX.
;
        CMP     RDHERE,TAKENX   ; prevent wrap
        BEQ     20$             ; if equal, no room in ring
        MOVB    (R0)+,@RDHERE   ; copy from temp buffer to ring buffer
        INC     RDHERE          ; point to the next location
        CMP     RDHERE,#RINGEND ; do we need to wrap the buffer?
        BLO     15$             ; if LO, no
        MOV     #RINGBF,RDHERE  ; else, next character
                                ; goes to beginning
15$:
        SOB     R1,10$          ; loop till all characters read
        BR      30$             ; done
20$:
        INC     WRAPFL          ; indicate that the task code is not
                                ; processing the data as fast as it's
                                ; coming in
30$:
        MOV     (SP)+,R1        ; restore R1
        MOV     (SP)+,R0        ; restore R0
        ASTX$S                  ; exit AST
```

```
;+
; PROCESS
;
; Input:        R0 (low byte) contains character
;               WRAPFL is set if data lost because coming in
;                       faster than going out
; Output:       ?
;-
PROCESS:
;
; Here's where the program deals with the input characters.
;
; Note that something must be done if WRAPFL is set.  If
; that is the case, it means data has been lost
; because the input is winning the race with this routine.
;
        CLC                     ; get another character
        RETURN

;+
; Static DPBs
;-
WAIT:   WTSE$   BUFEFN
CLEF:   CLEF$   BUFEFN
SETF:   SETF$   BUFEFN          ; indicate buffer has characters

ALUN:   ALUN$   TTLUN,TT,1      ; assign LUN
QIOATT: QIOW$   IO.ATA!TF.NOT!TF.XCC,TTLUN,QIOEFN,,IOSB,,<ASTADR>
QIORNE: QIOW$   IO.RNE!TF.TMO,TTLUN,QIOEFN,,IOSB,,<TMPBUF,TMPSIZ,0>

;+
; Data area
;-
RINGBF: .BLKB   200             ; ring buffer
RINGEND = .                     ; end of it

RDHERE: .WORD   0               ; next available location for input
TAKENX: .WORD   0               ; start of current input

IOSB:   .BLKW   2               ; I/O status block

TMPSIZ = 100
TMPBUF: .BLKB   TMPSIZ          ; temporary buffer for read QIOs

WRAPFL: .WORD   0               ; wraparound flag

        .END    START
```

APPENDIX A

DOCUMENTATION DIRECTORY

This appendix lists the manuals in the Tool Kit documentation set in alphabetical order, and provides an abstract of each. Note that the documentation set is organized so that related manuals are in the same volume (binder).

Software derived with little change from RSX-11M-PLUS layered products is documented in two manuals; the original RSX-11M/M-PLUS manual is included in the document set along with a supplement containing information specific to the Tool Kit version.

Note that the PRO/Tool Kit volume is provided only with the PRO/Tool Kit. You do not need this volume if you are using the Host Tool Kit.

- *CORE Graphics Library Manual* describes a general-purpose graphics subroutine library based on the ACM SIGGRAPH CORE Graphics Standard. The CORE Graphics Library provides a higher-level graphics programming interface than PRO/GIDIS.

- *FMS-11/RSX Software Reference Manual* describes how to develop FMS-11 applications on RSX-11M and RSX-11M-PLUS systems. Use it along with the *PRO/FMS-11 Documentation Supplement* and the *FMS-11/RSX Release Notes*.

- *Guide to Writing a P/OS I/O Device Driver and Advanced Programmer's Notes* defines Executive and I/O driver interface protocols, describes system I/O data structures, and suggests I/O driver routine coding procedures. It is written for the senior-level system programmer who is familiar with the hardware characteristics of both the Professional and the device that the user-written software supports. Unless explicitly noted otherwise, all information in this manual is subject to change without notice.

- *Host Tool Kit Installation Guide and Release Notes* contains instructions for installing the Tool Kit software on a host development system. It also provides information specific to the current release of the Host Tool Kit. You should disregard this manual if you are using the PRO/Tool Kit.

- *IAS/RSX-11 ODT Reference Manual* describes how to use the On-line Debugging Tool (ODT) to debug user task images. It is intended for all application developers. Use it along with the *IAS/RSX-11 ODT Supplement*. (Note that some high-level languages provide their own debugging facilities.)

- *IAS/RSX-11 ODT Supplement* describes the differences between IAS/RSX-11 ODT and ODT on the Professional, and should be used with the *IAS/RSX-11 ODT Reference Manual*.

- *IAS/RSX-11 System Library Routines Reference Manual* describes routines that were orignally written to provide commonly-needed capabilities for DIGITAL-supplied utilities. The routines are general enough to be used by most MACRO-11 programmers.

- *Introduction to DECnet* is an overview of the concepts and capabilities of DECnet networks. It describes the major network concepts behind all implementations of DECnet, defines specific network functions, and identifies the DECnet implementations that support each function.

- *PDP-11 MACRO-11 Language Reference Manual* describes how to use the MACRO-11 relocatable assembler to develop assembly language programs.

- *P/OS System Reference Manual* describes the P/OS Executive, the nucleus of the Professional Operating System.

- *Positional Device Interface Programmer's Manual* describes software that allows you to write applications that use positional devices, such as mice, bit pads, and touch screens.

- *PRO/DECnet Programmer's Reference Manual* discusses software requirements for creating PRO/DECnet applications. It reviews software design conventions that are critical to the early stages of program development and details network programming calls used in the creation of PRO/DECnet applications. It assumes that you have a working knowledge of networking concepts.

- *PRO/DECnet Tool Kit Installation Guide* details procedures for installing the PRO/DECnet Tool Kit either on an RSX-11M/M-PLUS or a VAX/VMS host system, or on a Professional computer. It also provides information on how to customize Professional systems as PRO/DECnet nodes.

- *PRO/DECnet Tool Kit Release Notes* provides information specific to the current release of the PRO/DECnet Tool Kit.

- *PRO/Document VDM Manual* describes the PRO/Document Virtual Device Metafile (VDM) interpreter. This interpreter provides a syntax for document control and text formatting. The manual provides a conceptual overview and reference information. It is intended for experienced programmers who are developing text editor and document display applications for the Professional.

- *PRO/FMS-11 Documentation Supplement* describes the differences between FMS/RSX and PRO/FMS, and should be used with the Tool Kit FMS-11/RSX documentation. It is intended for the application developer experienced with FMS-11.

- *PRO/GIDIS Manual* describes the General Image Display Instruction Set, a device-independent graphics interface specific to DIGITAL. PRO/GIDIS provides a low-level virtual device interface to the Profesional's graphics hardware. The GIDIS Call Interface (GIDCAL) allows you to uniformly access all supported output devices from high-level languages. The manual functions as a user's guide and reference manual. It is intended for developers with systems programming and graphics software experience. The sample programs are written in MACRO-11.

- *PRO/ReGIS Manual* describes DIGITAL's Remote Graphics Instruction Set. PRO/ReGIS runs under the control of PRO/Communications; it runs only in terminal emulation mode. You do not need graphic programming experience to use PRO/ReGIS.

- *PRO/RMS-11: An Introduction* introduces the concepts of RMS-11 record formats, file organizations, and record access modes. It does not provide reference or usage information, but should be read before the other RMS-11 documents.

- *PRO/RMS-11 Documentation Supplement* updates information contained in the three other RMS-11 manuals. The manual documents any differences between RMS-11 on the Professional and on RSX-11M/M-PLUS systems. It is intended for application developers who are familiar with RMS-11 facilities.

- *PRO/RMS-11 Macro Programmer's Guide* provides reference material about the macros and symbols that make up the interface between a MACRO-11 program and the RMS-11 operation routines. It is intended for application developers who are already familiar with RMS-11 facilities.

- *PRO/Tool Kit Command Language and Utilities Manual* describes the Digital Command Language (DCL) and program development utilities available on the PRO/Tool Kit.

- *PRO/Tool Kit Installation Guide and Release Notes* contains instructions for installing the PRO/Tool Kit software on a Professional. It also provides information specific to the current release of the PRO/Tool Kit.

- *RSX-11M/M-PLUS and Micro/RSX Task Builder Manual* describes the RSX-11M/M-PLUS Task Builder (TKB) upon which the Professional Appplication Builder (PAB) is based.

NOTE

The *Tool Kit User's Guide* (which you are reading) includes a chapter about task building. Use the *RSX-11M/M-PLUS and Micro/RSX Task Builder Manual* if you need more detailed information.

- *RSX-11M/M-PLUS RMS-11 User's Guide* is a guide to using RMS-11 in application programs written in either MACRO-11 or a high-level language. It is intended for application developers who are already familiar with RMS-11 facilities.

- *RSX-11M/M-PLUS RMS-11 Utilities Manual* describes the RMS-11 Utilities available to users of RMS-11 on an RSX-11M/M-PLUS host system. The PRO/Tool Kit implementation is a subset of those utilities and is described in the *PRO/Tool Kit Command Language and Utilities Manual*. This manual is intended for application developers who are using high-level languages and do not require or do not have access to the full set of RMS-11 capabilities.

- *Terminal Subsystem Manual* describes the hardware and software that control the Professional keyboard and video monitor (the functions typically performed by a video terminal). It documents how the Professional processes text in both native and terminal emulation modes.

- *Tool Kit Reference Manual* provides reference information about the following tools: Application Diskette Builder, PRO/Communications, Fast Install, File Control Services (FCS), Frame Development Tool (FDT), Installation Command Languages, MACRO-11 Assembler (PMA), POSRES User Interface Library, Print Services, PROSE Text Editor, and PRO/SORT.

- *Tool Kit User's Guide* is the manual you are reading. It describes each step of the program development cycle on the Professional computer using the Tool Kit. This manual is your primary source of general information about the Tool Kit.

APPENDIX B

GLOSSARY

application developer
The person who uses the Tool Kit to develop application programs for the Professional.

Application Diskette Builder (ADB)
The Tool Kit software component that builds a distribution diskette for an application by copying the appropriate files from disk. The program runs on the Professional.

application
The end result of the Tool Kit development cycle: a computer program that performs some useful service. In this manual, the term "application" often denotes a program and its related files.

checkpointing
The process by which the Executive makes physical memory available to higher-priority tasks by temporarily removing lower-priority tasks from memory and storing them on disk.

end user
The person who ultimately installs and runs your application.

Executive
See P/OS Executive.

Frame Development Tool (FDT)
A utility that creates menus, on-line help, and messages through an interactive session using forms. The displays are called frames, and frames are stored in frame files.

host

Another computer whose resources are available to your Professional via communication software, such as DECnet or the communications functions provided in P/OS. The term *Host Tool Kit* refers to the Tool Kit software running on a VAX/VMS or RSX-11M/M-PLUS operating system.

interactive program

A P/OS application that requires user interaction. On P/OS, only one interactive program can run at one time. An interactive program, however, can run at the same time as one or more noninteractive programs.

logical unit number (LUN)

There are two uses for this term:

- A logical unit number is number that indicates a particular unit of a device type. It is the nnn portion of the ddnnn: format of a physical device name.

- A *task* LUN appears in the ALUN$ and QIO$ directives; it is a number uniquely identifying an association between a physical device and your task.

noninteractive program

A program that runs primarily without user interaction, such as Print Services and File Transfer. On P/OS, multiple noninteractive programs can run concurrently with an interactive program.

object module

A file containing unrelocated binary instructions and data.

P/OS

The Professional operating system. P/OS is a single-user, real-time, multitasking system based on RSX-11M-PLUS.

P/OS Executive

The main component of the operating system. The Executive coordinates all activities and resources of the Professional. It provides task scheduling, interrupt processing, management of main memory, and coordination of I/O and file management facilities.

PRO/FMS-11 (Forms Management System)

A tool that performs screen management and data input using predefined forms.

P/OS Server
A component of P/OS that provides the following features:

- Ability to share resources, such as printers and storage media, among several Professionals.
- Access to all P/OS Hard Disk features without the need for a local hard disk.

P/OS stand-alone
A P/OS configuration that does not use a server.

PRO/RMS-11 (Record Management System)
A tool that provides an interface to the P/OS file system.

PRO/SORT
A tool that reorders data files according to control fields or key fields within the input data records.

RSX-11M-PLUS
A real-time, multitasking, operating system that makes use of the enhanced hardware features and memory of PDP-11 processors.

task
The fundamental executable program unit.

task builder
A tool (sometimes called a *linker*) that converts an object module into a task image by relocating code and data and resolving external references.

task image
A file that contains a loadable task in the form of absolute binary instructions and data.

terminal emulator
An application that allows the Professional to function as a terminal for the purpose of working on host systems.

Tool Kit
A set of software tools used to develop applications for the Professional computer.

user interface
The means by which an end user interacts with an application. The P/OS user interface consists of menus, on-line help, a message system, and standard use of function keys.

workstation

A Professional computer connected to a server in a P/OS Server environment.

INDEX