A/UX Development Tools

Final Draft

Developer Technical Publications

Printed in the United States of America.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
408-996-1010

Simultaneously published in the United States and Canada.

12/17/90 DTP Soft ©

Add applicable Apple registered trademarks (®'s) to the sixth paragraph

If your book mentions Apple trademarks (™s), add a credit paragraph after the sixth paragraph: "Name **and** Name **are trademarks of Apple Computer, Inc."**

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and tables

About This Book

This manual describes the A/UX® Developer's Tools, a collection of programs and utilities designed to help you create A/UX applications. It gives you the information you need to write applications and tools for the Apple® A/UX operating system. It assumes that you already know the C programming language and are familiar with the application development process.

What this manual contains

This manual contains these sections:

- This Preface describes the manual and directs you to other reference books with information about the C language and the Macintosh® and A/UX programming environments.
- Chapter 1, "A/UX Programming Environment," introduces the various environments your applications run in and discusses the tools available for use within those environments.
- Chapter 2, "Hybrid Applications," describes the characteristics of applications designed to take advantage of the strengths of both the A/UX and Macintosh features.
- Chapter 3, "Commando," documents how to create a Macintosh front-end for existing UNIX® applications without having to alter your source code.
- Chapter 4, "`dbx` Reference" discusses how to use the debugger provided with A/UX Developer's Tools.
- Chapter 5, "`c89` Command Syntax" is an overview of the command syntax of the compiler delivered with A/UX Developer's Tools.
- Chapter 6, "`as` Reference" discusses the features of the assembler delivered with A/UX Developer's Tools.
- Chapter 7, "The `ld` Loader" covers the loader provided with A/UX Developer's Tools.

Audience

This guide was written for three classes of program developers:

1. Those who have used C to create UNIX applications and want to add to the functionality of these applications by making calls to the Macintosh Toolbox creating, in effect, hybrid applications. This requires a working knowledge of Macintosh programming techniques. Chapters 1, 2, and 4–7 will be of most use to this group.
2. Those who integrate A/UX computers into already-existing networks and who want to take advantages of the powerful features inherent in A/UX. Those who simply want to add Macintosh front-ends to their programs without having to write new source code will find Chapter 3 of the most interest. Those wanting to call A/UX functions from Macintosh applications such as HyperCard will find Chapters 1, 2, and 4 of most use.
3. Developers who have created Macintosh applications and want to create specific device drivers to run under A/UX. This requires a detailed knowledge of UNIX programming techniques. Chapters 1, 2, and 4–7 will be of most use to this group.

Developers wishing to create applications specifically for the Macintosh Operating System may prefer to use the Macintosh Programmer's Workshop (MPW®), which is included in A/UX Developer's Tools. MPW contains a number of development tools specific to that environment.

Conventions used in this manual

This manual follows certain conventions regarding presentation of information. Words or terms that require special emphasis appear in specific fonts within the text. The following sections explain the conventions used in this manual.

Significant fonts

Code examples or words that might appear on your screen appear in `Courier` font. For example,

```
i = i + 1
```

The text shows `i = i + 1` in `Courier` typeface to indicate that it is sample code.

Words that you must replace with a value appropriate to a particular set of circumstances appear in *italics*. Using the example just described, a code sample might look like

```
i = i + constant
```

You would type in whatever value you wanted *constant* to be—for example,

```
i = i + 11091
```

Optional expressions are generally enclosed in square brackets. For example,

[*expression*]

indicates an optional expression.

New terms appear in **boldface** and are defined in the glossary of the manual.

Terminology

In A/UX manuals, a certain term can represent a specific set of actions. For example, the word *Enter* indicates that you type in an entry and press the RETURN key. If you were to see

Enter the following command: `whoami`

you would type `whoami` and press the RETURN key. The system would then respond by identifying your login name.

Here is a list of common terms and their corresponding actions.

Term	Action
Enter	Type in the entry and press the RETURN key.
Press	Press a *single* letter or key *without* pressing the RETURN key.
Type	Type in the letter or letters *without* pressing the RETURN key.
Click	Press and then immediately release the mouse button.
Select	Position the pointer on an item and click the mouse button.
Drag	Position the pointer on an icon, then press and hold down the mouse button while moving the mouse. Release the mouse button when you reach the desired position.

Choose Activate a command title in the menu bar. While holding down the mouse button, drag the pointer to a command name in the menu and then release the mouse button. An example is to drag the File menu down until the command name Open appears highlighted and then release the mouse button.

Aids to understanding

Look for these visual cues throughout the manual:

▲ **Warning** Warnings like this indicate potential problems. ▲

△ **Important** Text set off in this manner presents important information. △

◆ *Note:* Text set off in this manner presents notes, reminders, and hints.

Other reference material

You'll need to be familiar with these additional reference materials:

- *The C Programming Language.* (Second edition. Brian W. Kernighan and Dennis M. Ritchie. Prentice-Hall, 1988.) The standard reference book for the C language, rewritten for the then-proposed draft ANSI C.
- *A/UX Toolbox: Macintosh ROM Interface.* (Apple Computer, Inc., 1989.) Describes how to access and use the Macintosh ROM routines Available under A/UX.
- *A/UX Programming Languages and Tools*, Volumes 1 & 2. (Apple Computer, Inc., 1989.) Describes in detail the various tools available for program development under A/UX Version 2.0.
- *A/UX Programmer's Reference, Sections 2 and 3 (A–L).* (Apple Computer, Inc., 1989.) Describes in detail the various system calls and subroutines available under A/UX 2.0.
- *A/UX Programmer's Reference, Sections 3 (M–Z), 4, and 5.* (Apple Computer, Inc., 1989.) Describes in detail the various subroutines, file formats, and miscellaneous facilities available under A/UX 2.0.
- *A/UX ANSI C Reference.* (Apple Computer, Inc., 1991.) Describes the implementation of Apple's ANSI-compliant C compiler.

- *A/UX Porting Guide.* (Apple Computer, Inc., 1991.) Contains information for porting applications to run on A/UX. Describes the environment, compiler, and tools used under A/UX.

The following reference materials are included with A/UX Developer's Tools:

- *Macintosh Programmer's Workshop 3.0 Reference.* (Apple Computer, Inc., 1988.) Describes the Macintosh Programmer's Workshop, another development environment from Apple, including the editor, linker, and other important tools.
- *MacsBug Reference.* (Apple Computer, Inc., 1988.) Describes the MacsBug object-level debugger.
- *SADE Reference.* (Apple Computer, Inc., 1988.) Describes the SADE® source-level debugger.
- *ResEdit Reference.* (Apple Computer, Inc., 1988.) Describes the ResEdit™ resource editor.

You may also want to be familiar with these additional reference materials:

- *Human Interface Guidelines: The Apple Desktop Interface.* (Addison Wesley. 1987.) Describes the guidelines used to provide Macintosh computers a consistent interface.
- *The C Programming Language.* (First edition. Brian W. Kernighan and Dennis M. Ritchie. Prentice-Hall, 1978.) The standard reference book for the C language as originally defined. (The language it defines is called *K&R C.*)
- *C: A Reference Manual.* (Second edition. Samuel P. Harbison and Guy L. Steele, Jr. Prentice-Hall, 1987.) A standard reference book for the C language with the AT&T extensions used in most UNIX® operating system environments. The second edition contains a chapter on the then-proposed ANSI C standard.
- *American National Standard for Information Systems—Programming Language C.* (ANSI, 1989; document X3.159-1989.) This standard and its accompanying rationale define ANSI C, codifying many language extensions added since 1978. This book is herein referred to as the ANSI C standard. The language it describes is referred to as *ANSI C.*
- *Motorola MC68020 32-Bit Microprocessor User's Manual.* (Second edition. Prentice-Hall, 1985.) Describes the MC68020 processor in detail for hardware and software engineers.
- *MC68030 Enhanced 32-Bit Microprocessor User's Manual* (Second Edition , Englewood Cliffs, N. J.: Prentice-Hall, 1989). Describes the MC68030 processor in detail.
- *MC68881 Floating-Point Coprocessor User's Manual.* (Motorola, Inc., 1985.) Describes the instruction set and addressing conventions used by the MC68881 floating-point coprocessor, which is used in the Macintosh II.

- *MC68851 Paged Memory Management Unit User's Manual*, (Motorola, Inc., 1985.) Describes the instruction set and addressing conventions used by the MC68851 PMMU.

For more information

APDA® (Apple Programmers and Developers Association) offers worldwide access to a broad range of programming products, resources, and information for anyone developing on Apple platforms. You'll find the most current versions of Apple and third-party development tools, debuggers, compilers, languages, and technical references for all Apple platforms. To establish an APDA account, obtain additional ordering information, or find out about site licensing and developer training programs, please contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014-6299

800-282-2732 (United States)
800-637-0029 (Canada)
408-562-3910 (International)
Fax: 1-408-562-3971
Telex: 171-576
AppleLink® address: APDA

If you provide commercial products and services, please call 408-974-4897 for information on the developer support programs available from Apple.

Chapter 1 **A/UX Programming Environment**

This chapter describes the A/UX programming environment, and explains how the A/UX Developer's Tools enhance your ability to program applications for A/UX. The A/UX Developers' Tools combines the power of the UNIX® development environment with many features found in the Macintosh development environment.

A/UX as a UNIX operating system

A/UX fully complies with the System V Interface Definition (SVID) of the UNIX operating system. In addition, it meets the Federal Information Processing Standard (FIPS)-151 and IEEE 1003 Portable Operating System Interface for Computer Environments Full Use Standard (POSIX FUS)-1988. This adherence to standards provides developers with the knowledge that their compliant applications will usually run under A/UX with a simple recompile. The many features incorporated from Berkeley Software Distribution (BSD) 4.3 increase the ease of porting applications based on the Berkeley UNIX operating systems.

This combination of standards adherence and features extension thus gives you all the power inherent in a UNIX operating system—and preserves your investment if you've already developed UNIX application code. Indeed, the portability of code across the UNIX platforms is probably one of the reasons you chose UNIX in the first place.

A/UX as a Macintosh operating system

A/UX applications have access to the A/UX Toolbox, which allows A/UX applications to take advantage of Macintosh interface features such as a windowed environment and dialog box–based control of application parameters. A/UX Toolbox routines are particularly useful when you want to take advantage of the ease-of-use of the Macintosh desktop metaphor, but have a large amount of UNIX application code already written. Studies show that the Macintosh graphical interface is easier to learn than command-line interfaces. After users become accustomed to this interface, they view using a command line interface as less "elegant." The addition of Toolbox features can help make your application more accessible to the naive UNIX user. This, combined with the normal Macintosh applications that run under A/UX, can make your computing environment seamless, and thus more powerful and efficient.

A/UX Developer's Tools

The new set of tools provided with A/UX Developer's Tools include several new and enhanced tools for program development and porting in the A/UX environment

- `c89`, an ANSI-compliant C compiler. (new versions of the assembler and loader are included as well).
- A/UX system call libraries, a set of libraries that can be used to develop hybrid applications.
- `dbx`, a source-level debugger for A/UX and UNIX applications.
- X11R4, an implementation of the X Window System client architecture.

Additionally, A/UX Developer's Tools contains the following previously-released Macintosh development tools:

- Macintosh Programmer's Workshop (MPW™), Apple's fully integrated development environment, optimized for developing Macintosh applications. If your interest is primarily in these kinds of applications, please consult the *Macintosh Programmer's Workshop 3.1 Reference* included with A/UX Developer's Tools.
- MacsBug, an object-level debugger.
- ResEdit, a resource editor.
- SADE, a source-level debugger.

Installing A/UX Developer's Tools

The A/UX Developer's Tools software is contained on a 3 compact disc (CD) set. You must have a SCSI-based CD-ROM drive attached to your Macintosh system. Verify that you have the following CDs:

- A/UX Developer's Tools
- Macintosh Programmer's Workshop
- X Window System

Installation procedure

A/UX Developer's Tools contains an automated installation program. After you select the components you want to install, an MPW application called Installer takes care of transferring all files for you. As with any Macintosh application, you can press COMMAND-period to stop the installation at any time.

The Installer checks that you have enough disk space to complete the installation, then begins transferring the software. If there is not enough free space on your disk, the system warns you with an "Insufficient room to install" alert. Select Quit from the File menu of the Installer to stop the installation. You will need to remove some files from your disk to complete the installation.

Once the installer begins installing the software, it displays its progress in a window called Worksheet. Simultaneously it creates a file (named `errorList`) in the Installation Folder to track in detail any errors that occurred during the installation process. Once the installation has successfully completed, this file can be removed.

If the Installer needs to install software located on a separate CD, it ejects the current CD and prompts you for the required CD.

The general steps to install the software are:

1. Boot your A/UX system.
2. Drag the Installation folder from the A/UX Developer's Tools CD to your hard disk.
3. Launch the Installer application.
4. Select options to install the programs you want.
5. Quit the Installer.

△ **Important** In order for A/UX to be able to access the CD drive to install the software, the drive must be initialized. This is accomplished by booting the Macintosh Operating System from a system folder containing the CD driver. Therefore, you must start your Macintosh from a system folder that contains the driver for your CD drive. For many systems, this is the MacPartition volume. △

Preparing your system

1. Verify your CD drive is connected to your Macintosh and is powered on. Verify that there is *not* a CD in the drive.
2. Consult Table 1-1 to verify there is enough free space on your disk to install the desired software. This step is only needed for planning purposes; if there is not enough room to install an option, the installer reports this and stops.

■ **Table 1-1** Installation sizes

Option	Space required
A/UX System Calls	830 KB
`c89`	1,870 KB
`dbx`	??????? KB
A/UX C++	1,170 KB
X11R4	6,740 KB
MPW	8,930 KB
MPW C++	980 KB
MacsBug	310KB
ResEdit	770 KB
SADE	1,130 KB

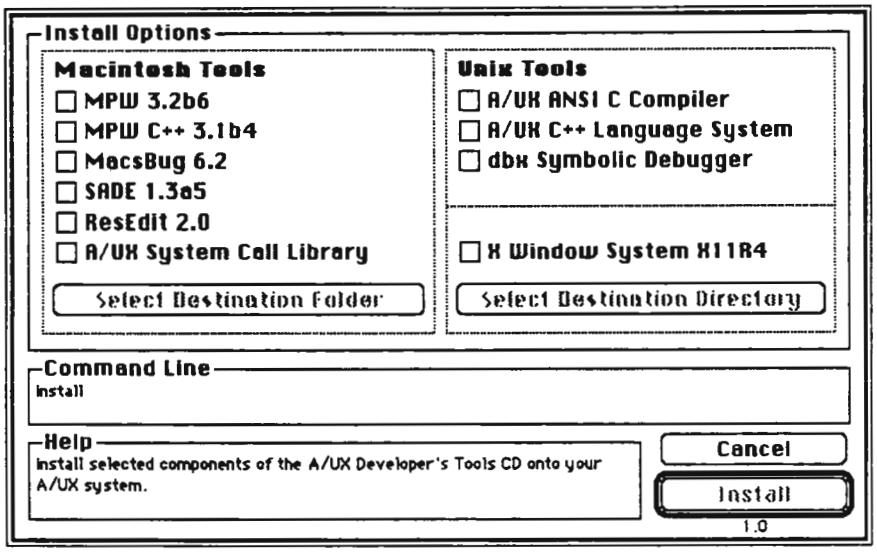
3. Start your Macintosh computer.
4. Boot the A/UX operating system.
5. Log on to your system as the root user.
6. Insert the A/UX Developer's Tools CD into your CD drive. The CD icon appears on your desktop and the disc's window opens on your desktop.
7. Drag the Installation Folder to any volume of your hard disk. The installation folder occupies about 800K of disk space, and must copied to a writable media to complete the installation.
8. Open the Installation Folder icon located on your hard disk. To avoid confusion, you may want to close the A/UX Developer's Tools window.

▲ **Warning** The Installer application overwrites files on your hard disk with compatible versions. Before running the application you may want to back up the programs `/bin/as` and `/bin/ld` and the directories `/lib`, `/usr/lib` and `/usr/include` ▲

Running the Installer

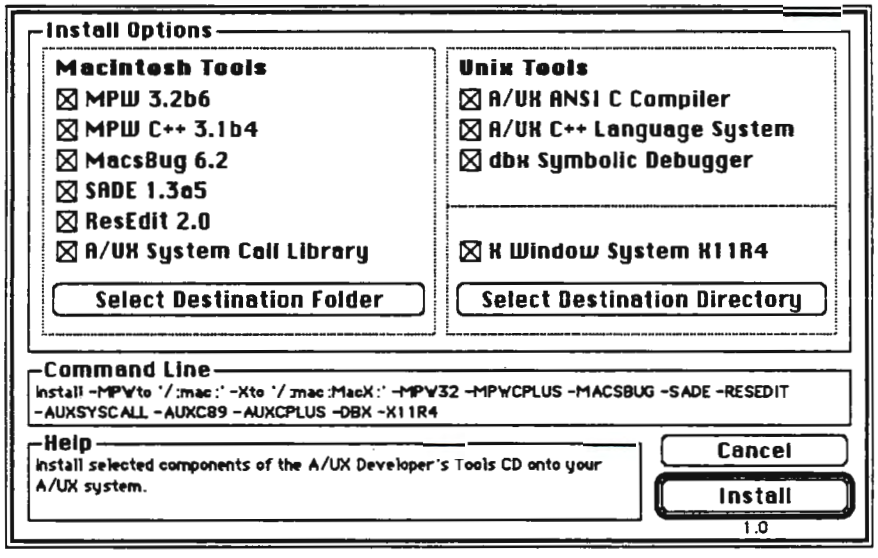
1. Open the Installer icon. A dialog box appears giving you choices of the various options you can install (see Figure 1-1).

■ **Figure 1-1** Installation dialog box



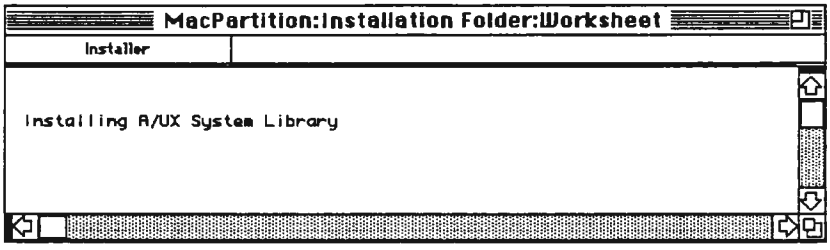
2. Select the checkboxes for the options you want to install. An installation dialog box with all the options selected is shown in Figure 1-2.

■ **Figure 1-2** Installation dialog box: All options selected



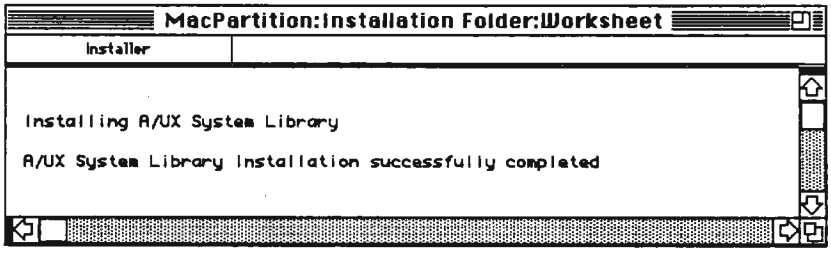
2. Select a destination folder if the options you want to install include any MPW Tools.
3. Select a destination directory if you want to install the X11R4 option.
4. Click the Install button. Installation time varies with the number and type of options selected.

- **Figure 1-3** Installation in process message



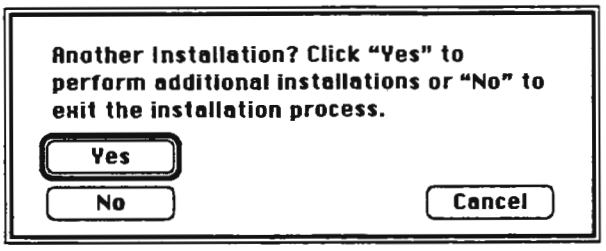
Once the installation is complete, the system informs you (see Figure 1-4).

- **Figure 1-4** Installation complete message



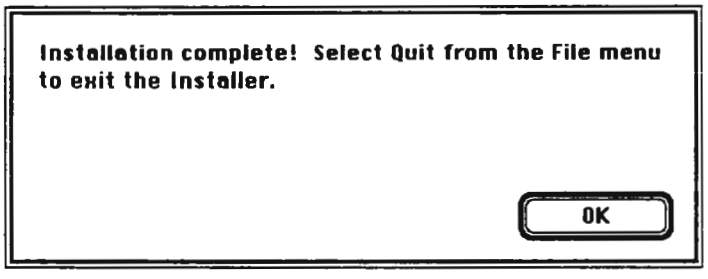
Once the system informs you of completion, the system asks if you want to perform another installation. (see Figure 1-5).

■ **Figure 1-5** Another Installation dialog box



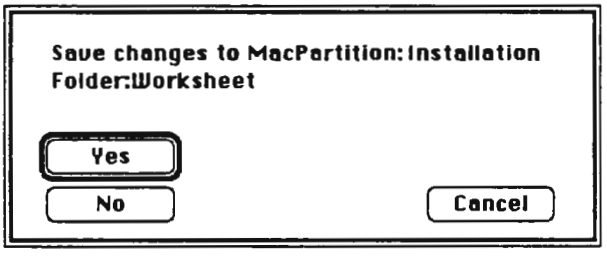
3. Click the No button. The system reminds you how to terminate the Installer. (see Figure 1-6).

■ **Figure 1-6** Installation complete dialog



4. Click the OK button.
5. Select Quit from the File menu to exit the installer. The system asks if you want to save the Worksheet file. (see Figure 1-7).

■ **Figure 1-7** Save Worksheet dialog box



6. Click the No button.

Installation of the A/UX Developer's Tools software is now complete.

Chapter 2 **Hybrid Applications**

This chapter describes how to create and debug hybrid applications—applications that combine the power of the UNIX® operating system with the ease of use inherent in the Macintosh user interface. This ability gives developers and system integrators tremendous flexibility in methods to increase the convenience and power of their applications.

Hybrid applications can be efficiently built within the MPW environment..Before you can effectively use the A/UX Toolbox to build hybrid applications from your UNIX applications, you must be familiar with Macintosh (event-driven) programming techniques.

Types of hybrid applications

A/UX Version 2.0 can run several classes of applications. In the first class are traditional UNIX applications that employ a terminal interface, or more recent UNIX applications that use the X Window System as a graphical interface. In the second class are Macintosh applications that are designed to work with the Macintosh Operating System, but can run unmodified on A/UX 2.0 provided they do not violate the Macintosh application guidelines for A/UX in *A/UX Toolbox: Macintosh ROM Interface.* the third class of applications are hybrid applications.

Hybrid applications are programs that employ techniques from both the UNIX and Macintosh application models. There are two basic types of hybrid applications. The first type is a UNIX application that uses the A/UX Macintosh Toolbox to provide an interface that has the Macintosh look and feel. This document refers to this type of hybrid application as a *UNIX hybrid application.* The A/UX CommandShell is an example of a UNIX hybrid application. The second type of hybrid application is a Macintosh application that makes UNIX system calls. This document refers to this type of hybrid application as a *Macintosh hybrid application.*

UNIX hybrid applications

UNIX hybrid applications are usually written in the C programming language and compiled with the A/UX C compiler. The A/UX `c89` C compiler has several language extensions that make it easier to create a UNIX hybrid application than was previously possible. UNIX hybrids are linked with the A/UX loader (`ld`) into a COFF (Common Object File Format) executable file that has special startup routines to attach the shared memory segment that is central to the A/UX MultiFinder® environment.

Programming guidelines for creating UNIX hybrid applications are given in *A/UX Toolbox: Macintosh ROM Interface* You can find additional details regarding the language extensions implemented in the A/UX ANSI C compiler in the *A/UX ANSI C Reference Manual.*

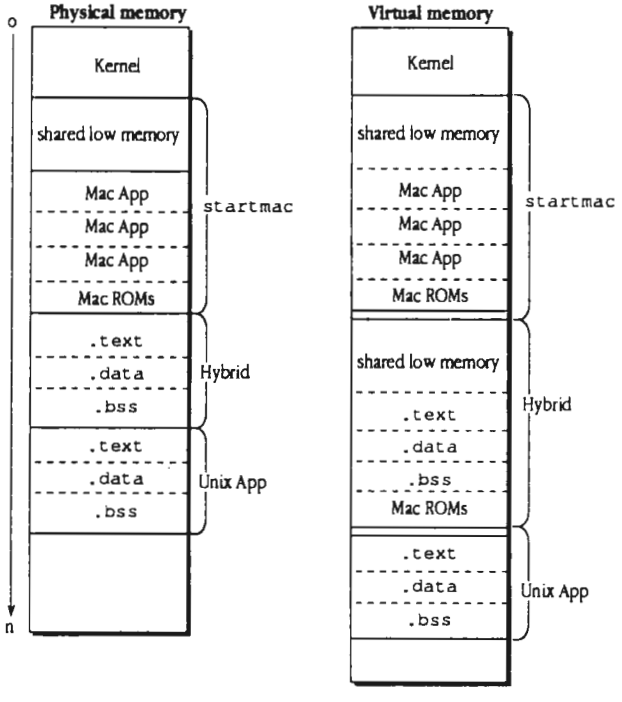
UNIX hybrid applications require a special debugging environment because of the location of programs in memory (see Figure 2-1). The A/UX Toolbox runs in **user space** in A/UX. This is a virtual, protected memory space that shares the system resources with all other processes running in user space. These processes are not allowed to access hardware directly. Instead, they must make a request to the A/UX kernel through a mechanism called a **system call** to deal with the hardware. The kernel, which runs in **system space**, then returns data, status, and other information back to the caller. The system call is a well-defined interface that gives UNIX systems some degree of application portability.

The physical memory layout of an A/UX system is shown on the left side of Figure 2-1. The kernel resides at the lowest addresses followed by a shared low memory segment. This segment includes the Macintosh ROMs, which contain all the Toolbox routines. the A/UX MultiFInder enfironment, running in a program called startmac, is next higher in addresses. The remainder of the physical memeory space is available for the text, data, and bss segments of whatever hybrid and UNIX applications may be running. These applications aresubject tothe normal swapping routines inherent in any UNIX system.

The virtual memory layout of an A/UX system is shown on the right side of Figure 2-1. All applications, except purely Macintosh applications, make requests for operating system services by means of system calls, shown in Figure 2-1 as arrows pointing into the kernel space. Each UNIX hybrid application has its own virtual mapping of the shared low memory and Macintosh ROMs.

The virtual mapping of low memory and ROM can make debugging somewhat complex. The `dbx` debugger contains some features that were specially designed to facilitate debugging UNIX hybrid applications. This chapter briefly describes those features and the general procedures for debugging UNIX hybrid applications. Chapter 4 contains a complete reference to `dbx`.

- **Figure 2-1** A/UX memory map



Macintosh hybrid applications

UNIX system calls are the basic application program interface (API) to the UNIX operating system. By careful use of UNIX system calls, a Macintosh hybrid application can access the powerful features of the UNIX operating system when running under A/UX. The A/UX Developer's Tools product contains a library of A/UX system calls that can be accessed from Macintosh applications. The system calls are designed to be called by programs written in the MPW C language, but other high-level language or assembly-language programs may also use the A/UX system call library if they can follow the MPW C calling conventions.

A special case of Macintosh hybrid application is a HyperCard stack that contains either XCMDs or XFCNs that make UNIX system calls. An XCMD is a code resource that implements a custom HyperCard command. An XFCN is a code resource that implements a custom HyperCard function. The difference between an XCMD and an XFCN is that an XFCN returns a value to the HyperCard stack that invoked it, whereas an XCMD performs an action without returning a result. XCMDs and XFCNs are created by writing the code for the command or function in a high-level language or in assembly language, compiling or assembling it, then linking it into code resource format. The code resource must be copied to either the HyperCard stack that calls the XCMD or XFCN, the HyperCard application itself, or any HyperCard stack that is visible in the stack hierarchy. The code resource is copied using the ResEdit program. More detailed information about creating XCMDs and XFCNs is contained in *Apple HyperCard Script Language Guide: The HyperTalk Language.*

▲ **Warning** The integration of the Macintosh API within the context of a multiprocess operating system like UNIX is made possible by a very precise balance of the requirements posed by each programming model. Developers of hybrid applications must take special care not to disturb this balance. Specific guidelines and areas to watch out for will be given in this document, but the developer should be prepared for the possibility of "spectacular crashes" during the initial phase of the development cycle. ▲

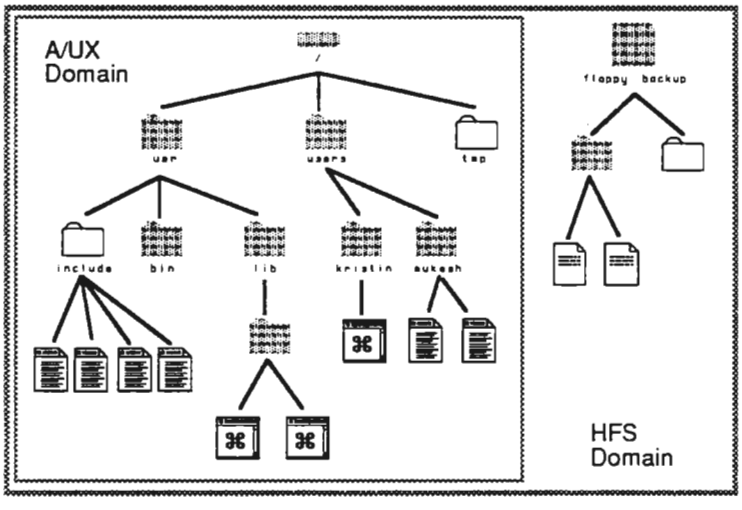
A/UX file systems

A/UX 2.0 supports both UNIX and Macintosh types of file systems. The UNIX file system is implemented by the A/UX kernel, and can be one of three types: UFS, SVFS, or NFS. All three share many common attributes, among them the convention of using a slash character (/) to separate directory components in a pathname and to indicate the root of the file system tree. These file systems are in consequence practically indistinguishable from an application program's perspective (See Figure 2-2). UNIX applications, UNIX hybrid applications, and Macintosh hybrid applications can access the UNIX file systems (shown as the A/UX Doamain in Figure 2-2) using UNIX system calls and library routines implemented with UNIX system calls. Macintosh applications can access the UNIX file systems as if they were Macintosh file systems contained on the volume named slash (/).

The Macintosh hierarchical file system (HFS) can reside on hard disks, floppy disks, and AppleShare® servers. A hard disk may be partitioned so that it contains a Macintosh file system and one or more UNIX file systems. The Macintosh file system, unlike UNIX file systems, uses a colon character (:) to separate directory (folder) components in a pathname. The root of a Macintosh file system hierarchy is indicated by the name of the volume on which it resides. Macintosh applications, Macintosh hybrid applications, and UNIX hybrid applications can access the Macintosh file systems using the A/UX Macintosh Toolbox.

◆ *Note:* UNIX applications cannot directly access Macintosh file systems.

■ **Figure 2-2** A/UX file systems



A/UX system call library

Developing Macintosh hybrid applications is simplified by the use of the A/UX system call library, which is part of the A/UX Developer's Tools product. Prior to the availability of this library, developers had to use assembly language to access A/UX system calls with the `trap` instruction. Using the A/UX system call library, developers writing Macintosh hybrid applications can access A/UX system calls from a high-level language in much the same way as if they were writing a UNIX application. The system call library is named `libaux_sys.o` and is an MPW format library file. Developers must include this library name in the MPW `Link` command that builds applications that use these system calls. For example:

```
Link -d -c 'MPS ' -t MPST  myapp.c.o  libaux_sys.o  ∂
      "{Libraries}"StdClib.o  "{Libraries}"Runtime.o
```

Because the MPW C programming environment implements many run time routines that have the same name as A/UX system calls, the A/UX system call library uses unique names for each system call.The naming convention adopted places the prefix `aux` prior to each system call name, for example, `auxread()`. This prefix differentiates MPW C run time routines and A/UX system calls.

- *Note:* All of the examples and procedures for creating Macintosh hybrid applications described in this document assume that you are using the MPW C development environment. If you are using another language or development environment you must adapt these procedures as if you were using an MPW C library from your environment.

A/UX system header files

Many A/UX system calls require arguments whose type or value is specified in A/UX system header files. The MPW C development environment also uses header files, many of which have the same name as the A/UX system header files. To avoid this ambiguity, you must give the full pathname of any A/UX system header file that is required in your application—for example,

```
#define _SYSV_SOURCE  1
#include </:usr:include:sys:errno.h>
#include <Stdio.h>
```

The first line in the example is needed to fool the A/UX system header file into thinking that it is being compiled by the A/UX C compiler. You could accomplish the same thing by compiling the program with the MPW C option to define a symbol, for example,

```
-d _SYSV_SOURCE
```

The second line in the example includes the system header file `errno.h` by giving the full pathname of the header file in Macintosh file system notation. The third line includes one of the MPW C header files; this is different than the A/UX header file `/usr/include/stdio.h`.

You should be aware that some A/UX system header files in the `/usr/include` directory are just pointers to the actual header file in the `/usr/include/sys` directory. When using these header files to create a Macintosh hybrid application, you must use the pathname of the actual header file, not the pointer to it. To determine if a header file is a pointer to another header file you must look at the header file to see if it includes another header file with the `#include` directive.

A/UX system calls and blocking

The A/UX MultiFinder environment is based on a special A/UX process called `startmac`. This process is in control of the virtual Macintosh environment, which is the default interface to A/UX from the system console. Because all Macintosh applications (including Macintosh hybrid applications) execute within this one process, it is critical that no single Macintosh application dominates the time slice allotted to the `startmac` process.

Macintosh applications interact with each other in a cooperative multitasking environment. What this means is that every Macintosh application will allow its execution to be interrupted before a significant period of time elapses. The mechanism for this interruption is the `WaitNextEvent` Macintosh Toolbox function. When a Macintosh application calls `WaitNextEvent`, it is a signal to MultiFinder that the application can be interrupted to give other applications executing in the `startmac` process an opportunity to perform some processing.

UNIX applications normally execute in a preemptive multitasking environment. This means that the operating system determines when to interrupt an executing process, and this may occur at any time the operating system chooses. Most often this happens when the application issues a system call that requires some external event to complete. The operating system will put that process to sleep until the external event has completed. In the meantime, other processes have an opportunity to execute. This behavior is referred to as *blocking*.

Blocking is normally not a problem for UNIX applications because they don't have anything to do until the system call completes anyway. For an application running within the `startmac` process, however, blocking prevents *all* of the Macintosh applications running in `startmac` from continuing. If the block remains in effect for more than a few seconds, the A/UX system console appears to be frozen.

To avoid this unpleasant situation, you must ensure that any A/UX system calls issued from a Macintosh hybrid application will complete in a brief period of time. For those system calls that have the potential to block, you must either know that blocking will not occur (for example, if you know that data is available when issuing an `auxread` system call), or else you must modify the behavior of the system call to eliminate blocking. You can change a system call's behavior by using the `auxfcntl()` system call to modify the status flags for a file descriptor to include the `O_NDELAY` flag. For more information about blocking and the `O_NDELAY` flag, see the `fcntl`(2) documentation in the *A/UX Programmer's Reference,* Section 2.

Listing of A/UX system calls

Tables 2-1 and 2-2 group the system calls available in the A/UX System call library according to their function.

This system calls in Table 2-1 control various input/output functions of the operating system.

■ **Table 2-1** Input/output system calls

`auxaccept`	`auxbind`	`auxconnect`
`auxgetsockopt`	`auxlisten`	`auxopen`
`auxpipe`	`auxread`	`auxreadv`
`auxrecv`	`auxrecvfrom`	`auxrecvmsg`
`auxselect`	`auxsend`	`auxsendmsg`
`auxsendto`	`auxsetsockopt`	`auxsocketpair`
`auxwrite`	`auxwritev`	

This system calls in Table 2-2 control various functions of the operating system.

■ **Table 2-2** Utilitysystem calls (Continued)

aux _exit	aux_ _sysm68K	aux_exit
auxaccess	auxcerror	auxchdir
auxchmod	auxchown	auxchroot
auxclose	auxcreat	auxdtablesize
auxdup	auxexec	auxexecl
auxexecle	auxexeclp	auxexecv
auxexecve	auxexecvp	auxexit
auxfchmod	auxfchown	auxfcntl
auxflock	auxfsmount	auxfstat
auxfstatfs	auxfsync	auxftruncate
auxgetcompat	auxgetdirent	auxgetdomain
auxgetegid	auxgetenv	auxgeteuid
auxgetgid	auxgetgroups	auxgethostid
auxgethostnam	auxgetitimer	auxgetpeername
auxgetpid	auxgetppid	auxgetsockname
auxgettod	auxgetuid	auxlink
auxlocking	auxlseek	auxlstat
auxmkdir	auxmsgsys	auxnfs_getfh
auxnfssvc	auxreadlink	auxrename
auxrmdir	auxsemsys	auxshmsys
auxsigcall	auxsigcode	auxsignal
auxsigvec	auxsocket	auxstat
auxstatfs	auxstime	auxsymlink
auxsync	auxsyscall	auxtime
auxtimes	auxtruncate	auxumask
auxumount	auxuname	auxunlink
auxunmount	auxustat	auxutime
auxuvar	auxwait	auxwait3
GetAUXErrno	SetAUXErrno	

Description of A/UX system calls

The following section describes all of the system calls available in the A/UX System call library, listed alphabetically. Unless otherwise specified, the usage and functionality of each system call is as described for its counterpart in the *A/UX Programmer's Reference,* Section 2.

- `auxaccept`—accept a connection on a socket

```
#include </:usr:include:sys:types.h>
#include </:usr:include:sys:socket.h>
int auxaccept(int s, struct sockaddr *addr, int *addrlen)
```

- `auxaccess`—determine accessibility of a file

```
#include </:usr:include:unistd.h>
int auxaccess(char *path, int amode)
```

- `auxbind`—bind a name to a socket

```
#include <sys/types.h>
#include <sys/socket.h>
int auxbind(int s, struct sockaddr *name, int namelen)
```

- `auxcerror`—need description
- `auxchdir`—change working directory

```
int auxchdir(char *path)
```

- `auxchmod`—change mode (permissions) of a file

```
#include <sys/types.h>
#include <sys/stat.h>
int auxchmod(char *path, mode_t mode)
```

- `auxchown`, `auxfchown`—change owner and group of a file

```
#include <sys/types.h>
int auxchown(char *path, uid_t owner, gid_t group)
int auxfchown(int fd, owner, group)
```

- `auxchroot`—change root directory

```
int auxchroot(char *path)
```

- `auxclose`—close a file descriptor

```
int auxclose(int fildes)
```

- `auxconnect`—initiate a connection to a socket

```
#include <sys/types.h>
#include <sys/socket.h>
int auxconnect(int s, struct sockaddr *name, int namelen)
```

- `auxcreat`—create a new file or rewrite an existing file

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
int auxcreat(char *path, mode mode)
```

- `auxdtablesize`—need description
- `auxdup`—duplicate a descriptor

```
int auxdup(int oldd)
```

- `auxexec`, `auxexecl`, `auxexeclp`, `auxexecle`, `auxexecv`, `auxexecve`, `auxexecvp`—execute a file

```
int auxexecl(char *path, *arg0, *arg1, ..., *argn)
int auxexecv(char *path, *argv[])
int auxexecle(char *path, *arg0, *arg1, ..., *argn, *envp[])
int auxexecve(char *path, *argv[], *envp[])
int auxexeclp(char *file, *arg0, *arg1, ..., *argn, 0)
int auxexecvp(char *file, *argv[], extern char **environ)
```

- `auxexecl`—see `auxexec`
- `auxexeclp`—see `auxexec`
- `auxexecle`—see `auxexec`
- `auxexecv`—see `auxexec`
- `auxexecve`—see `auxexec`
- `auxexecvp`—see `auxexec`
- `auxexit`, `aux_exit`—terminate process

```
void auxexit(int status)
void aux _exit(int status)
```

- `auxfchmod`—need description
- `auxfchown`—see `auxchown`
- `auxfcntl`—file control

```
#include <sys/types.h>
#include <fcntl.h>
int auxfcntl(int fildes, cmd, arg)
```

- `auxflock`—apply or remove an advisory lock on an open file

```
#include <sys/file.h>
auxflock(int fd, operation)
```

- `auxfsmount`—mount a network file system (NFS)

```
#include <sys/mount.h>
int auxfsmount(int type, char *dir, int flags, caddr_t data)
```

- `auxfstat`—see `auxstat`
- `auxfstatfs`—see `auxstatfs`
- `auxfsync`—synchronize a file's in-core state with that on disk

```
int auxfsync(int fd)
```

- `auxftruncate`—see `auxtruncate`
- `auxgetcompat`—need description
- `auxgetdirent`—get directory entries .

```
#include <sys/types.h>
#include <sys/dir.h>
int auxgetdirent(int d, char *buf, int nbytes, long *basep)
```

- `auxgetdomain`—get the domain name of the current network domain

```
int auxgetdomain(char *name, int namelen)
```

- `auxgetegid`—see `auxgetuid`
- `auxgetenv`—need description
- `auxgeteuid`—see `auxgetuid`
- `auxgetgid`—see `auxgetuid`
- `auxgetgroups`—get group access list

```
#include <sys/param.h>
int auxgetgroups(int gidsetlen, *gidset)
```

- `auxgethostid`—get unique identifier of current host

```
int auxgethostid()
```

- `auxgethostnam`—get name of current host

```
int auxgethostnam(char *name, int namelen)
```

- `auxgetitimer`—get value of interval timer

```
#include <sys/time.h>
auxgetitimer(int which, struct itimerval *value)
```

- `auxgetpeername`—get name of connected peer

```
int auxgetpeername(int s, struct sockaddr *name, int *namelen)
```

- `auxgetpid`, `auxgetppid`—get process or parent process IDs

```
#include <sys/types.h>
pid_t auxgetpid()
```

```
pid_t auxgetppid()
```

- `auxgetppid`—see `auxgetpid`
- `auxgetsockname`—get socket name

```
int auxgetsockname(int s, struct sockaddr *name, int *namelen)
```

- `auxgetsockopt`, `auxsetsockopt`—get and set options on sockets

```
#include <sys/types.h>
#include <sys/socket.h>
int auxgetsockopt(int s, level, optname, char *optval, int *optlen)
int auxsetsockopt(int s, level, optname, char *optval, int *optlen)
```

- `auxgettod`—get time and date

```
#include <sys/time.h>
int auxgettod(struct timeval *tp, struct timezone *tzp)
```

- `auxgetuid`, `auxgeteuid`, `auxgetgid`, `auxgetegid`—get real and effective user IDs and group IDs

```
#include <sys/types.h>
uid_t auxgetuid()
uid_t auxgeteuid()
uid_t auxgetgid()
uid_t auxgetegid()
```

- `auxlink`—link to a file

```
int auxlink(char *path1, *path2)
```

- `auxlisten`—listen for connections on a socket

```
auxlisten(int s, backlog)
```

- `auxlocking`—provide exclusive file regions for reading or writing

```
int auxlocking(int fildes, int mode, int size)
```

- `auxlseek`—move read/write file pointer

```
#include <sys/types.h>
#include <unistd.h>
off_t auxlseek(int fildes, off_t offset, int whence)
```

- `auxlstat`—see `auxstat`
- `auxmkdir`—make a directory file

```
int auxmkdir(char *path, int mode)
```

- `auxmknod`—make a directory, or a special or ordinary file

```
int auxmknod(char *path, int mode, dev)
```

- `auxmsgsys`—need description
- `auxnfs_getfh`—get a file handle

```
#include <rpc/types.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <nfs/nfs.h>
int auxnfs_getfh(int fildes, fhandle_t *fhp)
```

- `auxnfssvc`—NFS daemon

```
int auxnfssvc(int sock)
```

- `auxopen`—open for reading or writing .

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int auxopen(char *path, int oflag[, int mode])
```

- `auxpipe`—create an interprocess channel

```
int auxpipe(int fildes [2])
```

- `auxread`, `auxreadv`—read from file

```
#include <sys/types.h>
#include <sys/uio.h>
int auxread(int fildes, char *buf, unsigned nbytes)
int auxreadv(int fildes, struct iovec *iov, int iovcnt)
```

- `auxreadlink`—read value of symbolic link

```
int auxreadlink(char *path, *buf, int bufsiz)
```

- `auxreadv`—see `auxread`
- `auxrecv`, `auxrecvfrom`, `auxrecvmsg`—receive a message from a socket

```
#include <sys/types.h>
#include <sys/socket.h>
int auxrecv(int s, char *buf, int len, flags)
int auxrecvfrom(int s, char *buf, int len, flags, struct sockaddr
*from, int *fromlen)
int auxrecvmsg(int s, struct msghdr msg[], int flags)
```

- `auxrecvfrom`—see `auxrecv`
- `auxrecvmsg`—see `auxrecv`
- `auxrename`—change the name of a file

```
int auxrename(char *from, *to)
```

- `auxrmdir`—remove a directory file

```
int auxrmdir(char *path)
```

- `auxselect`—synchronous I/O multiplexing

```
#include <sys/time.h>
int auxselect(int nfds, *readfds, *writefds, *execptfds, struct timeval
*timeout)
```

- `auxsemsys`—need description
- `auxsend`, `auxsendto`, `auxsendmsg`—send a message from a socket

```
#include <sys/types.h>
#include <sys/socket.h>
int auxsend(int s, char *msg, int len, flags)
int auxsendto(int s, char *msg, int len, flags, struct sockaddr *to,
int tolen)
int auxsendmsg(int s, struct msghdr msg[], int flags)
```

- `auxsendmsg`—see `auxsend`
- `auxsendto`—see `auxsend`
- `auxsetsockopt`—see `auxgetsockopt`
- `auxshmsys`—need description
- `auxsigcall`—need description
- `auxsigcode`—need description
- `auxsignal`—need description
- `auxsigvec`—optional BSD-compatible software signal facilities

```
#include <signal.h>
struct auxsigvec {
    int (*sv_handler)();
    int sv_mask;
    int sv_flags;
};
int auxsigvec(int sig, struct sigvec *vec, *ovec)
```

- `auxsocket`—create an endpoint for communication

```
#include <sys/types.h>
#include <sys/socket.h>
int auxsocket(int af, type, protocol)
```

- `auxsocketpair`—need description
- `auxstat`, `auxfstat`, `auxlstat`—get file status

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
int auxstat(char *path, struct stat *buf)
int auxfstat(int fildes, struct stat *buf)
int auxlstat(char *path, struct stat *buf)
```

- `auxstatfs`, `auxfstatfs`—get file system statistics

```
#include <sys/types.h>
#include <sys/vfs.h>
int auxstatfs(char *path, struct statfs *buf)
int auxfstatfs(int fildes, struct statfs *buf)
```

- `auxstime`—set time

```
int auxstime(long *tp)
```

- `auxsymlink`—make a symbolic link to a file

```
int auxsymlink(char *name1, *name2)
```

- `auxsync`—update superblock

```
void auxsync()
```

- `auxsyscall`—need description
- `auxtime`—get time

```
#include <time.h>
time_t auxtime((long*)0)
time_t time(tloc);
time_t (*tloc);
```

- `auxtimes`—get process and child process times

```
#include <sys/types.h>
#include <sys/times.h>
clock_t auxtimes(struct tms *buffer)
```

- `auxtruncate`, `auxftruncate`—truncate file to a specified length

```
int auxtruncate(char *path, int length)
int auxftruncate(int fd, length)
```

- `auxumask`—set and get the file creation mask

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t auxumask(cmask)
mode_t cmask)
```

- `auxumount`—unmount a file system

```
int auxumount(char *spec)
```

- `auxuname`—get name of current system

```
#include <sys/utsname.h>
int auxuname(struct utsname *name)
```

- `auxunlink`—remove directory entry

```
int auxunlink(char *path)
```

- `auxunmount`—remove a file system

```
auxunmount(char *name)
```

- `auxustat`—get file system statistics

```
#include <sys/types.h>
#include <ustat.h>
int auxustat(int dev, struct ustat *buf)
```

- `auxutime`—set file access and modification times

```
#include <sys/types.h>
#include <utime.h>
int auxutime(char *path, struct utimbuf *times)
```

- `auxuvar`—return system-specific configuration information

```
#include <sys/var.h>
int auxuvar(struct var *v)
```

- `auxwait`—wait for a child process to stop or terminate

```
int auxwait(int *stat_loc)
int auxwait((int*)0)
```

- `auxwait3`—nonblocking wait for a child process to stop or terminate

```
#include <sys/wait.h>
int auxwait3(union wait *status, int options, 0)
```

- `auxwrite`, `auxwritev`—write on a file

```
#include <sys/types.h>
#include <sys/uio.h>
int auxwrite(int fildes, char *buf, unsigned nbytes)
int auxwritev(int fildes, struct iovec *iov, int ioveclen)
```

- `auxwritev`—see `auxwrite`
- `aux_exit`—see `auxexit`
- `aux__sysm68K`—need description
- `GetAUXErrno`—return the A/UX `errno` value

```
#include </:usr:include:sys:errno.h>
int GetAUXErrno()
```

- `SetAUXErrno`—set the A/UX `errno` value

```
#include </:usr:include:sys:errno.h>
void SetAUXErrno(int value)
```

Additional library routines

Creating a Macintosh hybrid application can be greatly simplified by using the additional library routines described in this section. These routines were developed to provide a higher level of functionality than the basic system calls provide. The source code for these library routines is also provided with A/UX Developer's Tools so that you can customize them if your needs require.

`auxfork_pipe`

The `auxfork_pipe()` function will perform a `fork` system call, set up optional input/output pipes between the parent and child processes, and call a procedure to perform the final processing for the child process. The `auxfork_pipe()` function returns a Macintosh memory handle if the function completed successfully; otherwise it returns the value `NULL`. The function is defined as

```
#define TOCHILD  0
#define TOPARENT 1
#define PID      2
#include <Memory.h>
Handle auxfork_pipe(int toparent, int tochild,
                    void (*childtask)(), void *childarg)

int cleanup_auxfork_pipe(Handle globals)
```

The `toparent` argument should have a nonzero value if you want the parent process to be able to read the standard output and standard error messages that the child process may write. The `tochild` argument should have a nonzero value if you want the parent process to be able to write messages that the child process will read from standard input.

The `childtask` argument is a function pointer that is used to call a function from the child process after the pipes have been set up. The function that `childtask` points to *must* call either one of the `auxexec()` system calls or the `auxexit()` system call. If it does not, two copies of the A/UX Macintosh environment are left active, rendering the A/UX Macintosh environment inoperable. The `childarg` argument allows you to pass one generic pointer argument to the `childtask` function. Multiple arguments to `childtask` can be encoded as a list pointed to by `childarg`, which the `childtask` function can then decode into individual arguments.

The handle returned by `auxfork_pipe()` on successful completion points to an area of memory that contains several global values that are needed for subsequent functions. The `auxcleanup_fork_pipe()` function must be called when your application has completed interacting with the child process and is ready to have the child process terminate. This function closes pipes, releases memory, and waits for the child to terminate.

The constants `TOCHILD`, `TOPARENT` and `PID` defined in the `auxfork_pipe()` function are used to access some of the global values that `auxfork_pipe` stores in the memory pointed to by the handle. The following is an example of how to access these values:

```
Handle global_handle;
int    *globals;
int    inpipe,outpipe,childpid;

global_handle = auxfork_pipe(1,1,childtask,childarg);
if ( global_handle ) {
   globals =  (int *)*global_handle;
   inpipe =   globals[TOPARENT];
   outpipe =  globals[TOCHILD];
   childpid = globals[PID];
}
```

The variable `inpipe` set by this procedure holds the file descriptor that the parent process can use to read the standard output and standard error data produced by the child process. The variable `outpipe` holds the file descriptor that the parent process can use to write to the standard input of the child process. The variable `childpid` holds the UNIX process ID of the child process.

- *Note:* Do not call `auxcleanup_fork_pipe()` until you are sure that the child process is ready to terminate. The call to `auxcleanup_fork_pipe()` will not return until the child process terminates, and the A/UX Macintosh environment will not be usable in the interim. If you want to start a background process, you should have `auxfork_pipe()` start a process that activates the background process and then terminate so that `auxcleanup_fork_pipe()` can be successfully called after calling `auxfork_pipe()`.

auxsystem

The `auxsystem()` function executes an A/UX command and sends all of the standard output and standard error output from the command to the standard output of the calling program. This function works only if called by an MPW program that is running with the standard I/O facilities of the MPW shell environment. The function is defined as

```
int auxsystem(char *command)
```

The value returned by `auxsystem` is the status code with which the A/UX command exited.

auxfgets

The `auxfgets()` function simulates the behavior of the UNIX library function `fgets()`. The argument `buf` points to an area of memory where the text line will be stored, the argument `count` indicates the size in bytes of that memory area, and the argument `file` is the file descriptor from which to read. The file descriptor will usually be the inpipe file descriptor, which is obtained with the `auxfork_pipe()` function described previously. The function is defined as

```
char *auxfgets(char *buf, int count, int file)
```

The function reads from the given file descriptor until it either fills the buffer, encounters a new-line character, an error condition occurs, or the timeout value is exceeded (the timeout value is the number of retries; 0 means there is no timeout). If read, the terminating new-line character will be placed in the buffer. Space is reserved in the buffer for a null byte to indicate the end of the input.

- *Note:* See the information on system calls that block in the section "A/UX System Calls and Blocking" earlier in this chapter.

HyperCard XCMDs and XFCNs

One of the more elegant uses of the A/UX system call library is to create HyperCard XCMDs and XFCNs to allow HyperCard stacks to access UNIX. This special type of Macintosh hybrid application is easier to create because of the power and flexibility of the HyperTalk® scripting language.

A sample Macintosh hybrid application using HyperCard

The ability to issue A/UX system calls from HyperCard external functions (XFCNs) is an important use of the A/UX system call library. A sample application (in HyperCard an application is referred to as a *stack*) is included with A/UX Developer's Tools. This sample application implements a UNIX mail reader via HyperCard. To give you a better understanding of how to create a HyperCard stack that uses A/UX system calls the following sections examine how this sample application is implemented. All of the source code for the stack and its XFCNs is included, so you can take this example and customize it to your requirements. (For more information about writing XFCNs and HyperCard stacks, refer to *Apple HyperCard Script Language Guide: The HyperTalk Language.*)

This application provides a graphical interface to the UNIX command `mailx`. Normally this command is run from a UNIX shell in character mode. The program displays lines of text to the user, and the user can enter text commands to the program when prompted to do so. This sample application acts as a front-end to the `mailx` program, so that the user can interact with `mailx` in the Macintosh style of direct manipulation by pointing with the mouse and clicking a mouse button.

The HyperCard XFCNs

The HyperCard stack must be able to perform four basic operations in order to implement this type of front-end application. In this section, these four operations are described in detail because they are building blocks that you can use to create your own Macintosh hybrid applications using HyperCard.

The XFCN `forkpipexfcn`

First, the application must be able to initiate execution of the UNIX program (referred to as the *child process*) and establish communication channels (referred to as *pipes*) between HyperCard (the *parent*[1]) and the child process. In this case, the UNIX program is named `/usr/ucb/mailx`. This is accomplished using the XFCN named `forkpipexfcn`, which is based on the function `auxfork_pipe()`, contained in the A/UX system call library (`libaux_sys.o`).

This XFCN accepts one parameter, which is a string containing the name of the UNIX command to be executed. The following section of the XFCN `forkpipexfcn` checks to see that exactly one parameter has been passed to it from the HyperCard stack. It fetches that parameter and places it into the character pointer named `command`, and then passes that pointer as the last parameter to the `auxfork_pipe()` function. The parameters `toparent` and `tochild` are flags that instruct `auxfork_pipe()` to set up pipes for communication from the child to the parent, and from the parent to the child. The parameter `childtask` is a function pointer that will be called by the child process to execute the UNIX command that is specified by the parameter `command`.

```
if ( paramPtr->paramCount != 1 ) global_handle = NULL;
else {
      command = *(paramPtr->params[0]);
      global_handle =
            auxfork_pipe(toparent,tochild,childtask,command);
};

paramPtr->returnValue = hexhandle(global_handle,8);
```

The value returned by `auxfork_pipe()` is a handle that points to a memory area that contains information about the child process and the pipes that have been set up to communicate with it. This information is required by the other XFCNs that will be used to communicate with the child process. The function `hexhandle()` converts the handle address into a hexadecimal string that is stored in another handle to be returned to HyperCard as the result of the XFCN.

[1]In reality, the parent process is `startmac`, the A/UX MultiFinder environment. HyperCard is one of possibly many Macintosh applications that may be running within the `startmac` process under the control of MultiFinder. For the purposes of this discussion, the term *parent* refers to the specific Macintosh application that initiated execution of the child process.

The function `childtask()` is needed to make `auxfork_pipe()` generic. It makes the A/UX system call `auxexecl()` to turn the child process, which initially is a copy of the `startmac` process, into the UNIX command to be executed. It is *very* important that this function do nothing more than perform an `exec` system call by calling `auxexecl()` or one of the other A/UX system calls that `exec` a command. (If an `exec` call is not immediately made, there are two copies of the `startmac` process, a situation that will quickly crash the system.) The call to `aux_exit()` that follows the call to `auxexecl()` is required in case `exec` failed for some reason.

```
void childtask(command)
      char *command;
{
      (void) auxexecl(command, 0);
      (void) aux_exit(127);
}
```

If the UNIX command to be executed requires arguments, you can modify this XFCN so that the parameter `command` that is passed to `childtask()` points to a list of arguments to be included in the `exec` system call.

The XFCNs `fgetsxfcn` and `fgetfxfcn`

The next operation is to read data from the child process. The XFCN `forkpipexfcn` set up two pipes for communicating with the child process. One of these pipes reads data that the child process writes to its standard output or standard error streams.

This is accomplished using the XFCNs `fgetsxfcn` and `fgetfxfcn`. These XFCNs are based on the function `auxfgets()`, which is contained in the A/UX system call library (`libaux_sys.o`). Here is a listing for the main code to the XFCN `fgetsxfcn`.

```
void do_xcmd(paramPtr)
XCmdPtr     paramPtr;
{
      char        buf[256];
      int         cnt,timeout,toparent;
      Handle      input,global_handle;
      int         **globals;

      if ( paramPtr->paramCount < 1 ) return;

      global_handle = (Handle) handlehex(*(paramPtr->params[0]));
      if ( paramPtr->paramCount > 1 )
            timeout = (int) timedec(*(paramPtr->params[1]));
      else timeout = 0;
```

```
        globals = (int **)*global_handle;
        toparent = (int) globals[TOPARENT];

        if ( (char *)auxfgets(buf,256,toparent,timeout) ) {
              cnt = strlen(buf);
              if ( cnt && (buf[cnt-1] == '\n') ) {
                    if ( cnt == 1 ) buf[0] = ' ';
                    else buf[--cnt] = '\0';
              };
        }
        else cnt = 0;

        input = NewHandle(cnt+1);
        strcpy(*input,buf);
        paramPtr->returnValue = input;
}
```

This XFCN accepts either one or two parameters. The first is the handle that was returned as the value of the XFCN `forkpipexfcn`. The second parameter is an optional timeout value that is used to determine how many times to try reading data from the child before giving up. The timeout value should be adjusted (empirically) to allow enough time for the child process to write a line of data to the pipe.

The code for this XFCN checks the parameter count, converts the first parameter from a hexadecimal string to a binary handle address, and converts the second parameter, if present, from a decimal string to a binary integer. If the second parameter is absent the value 0 is used; it denotes no timeout (that is, continue trying to read data until successful).

△ **Important** You should omit the timeout value only if you are absolutely certain that the child will write data to the pipe. If you omit the timeout value and the child fails to write data to the pipe, the XFCN will not complete and the entire A/UX MultiFinder environment will lock up. △

After converting the parameters, the XFCN fetches the file descriptor for the pipe that reads data from the child and calls the `auxfgets()` function from the A/UX system call library using the pipe file descriptor and the timeout value passed to it. The `auxfgets()` function reads from the pipe until it encounters a newline character (the end of a UNIX text record) or until the number of retries specified by the timeout value is reached. The `auxfgets()` function replaces the trailing newline character with a carriage return character for compatibility with the Macintosh environment.

The XFCN then computes the length of the read data and strips off the trailing carriage return character if it exists. The maximum length of the data read is 256 bytes. The data is then stored in memory pointed to by a new handle and returned to HyperCard as the return value of the XFCN. If no data was read, the XFCN will return the value empty to HyperCard. If only a newline character was read, a single blank will be returned to HyperCard to distinguish this from the condition where no data was read.

This XFCN `fgetfxfcn` is a variation on the XFCN `fgetsxfcn`. It reads multiple lines of text from the child process until the timeout value is exceeded while attempting to read the next line. This XFCN returns a value consisting of 0 or more lines of text separated by the carriage return character. The value returned to HyperCard by `fgetfxfcn` is suitable for placing into a multiline HyperCard text field. The following is the source code for this XFCN.

```
void do_xcmd(paramPtr)
XCmdPtr     paramPtr;
{
      char        buf[256],eos='\0';
      int         cnt,timeout,toparent;
      Handle      input,global_handle;
      int         **globals;

      if ( paramPtr->paramCount < 1 ) return;

      global_handle = (Handle) handlehex(*(paramPtr->params[0]));
      if ( paramPtr->paramCount > 1 ) timeout = (int)
            timedec(*(paramPtr->params[1]));
      else timeout = 0;

      globals = (int **)*global_handle;
      toparent = (int) globals[TOPARENT];
      input = NewHandle((long) 0);

      while ( (char *)auxfgets(buf,256,toparent,timeout) ) {
            cnt = strlen(buf);
            PtrAndHand (buf, input, cnt);
      };

      PtrAndHand (&eos, input, 1);
      paramPtr->returnValue = input;
}
```

The XFCN `writexfcn`

The next operation required is to write data to the child process. The second pipe set up by `forkpipexfcn` is used to write data from the parent to the child process.

This is accomplished using the XFCN named `writexfcn`. This XFCN is based on the function `auxwrite()`, which is contained in the A/UX system call library (`libaux_sys.o`). Here is a listing of the main code for this XFCN.

```
void do_xcmd(paramPtr)
XCmdPtr     paramPtr;
{
      char          *buf;
      int           cnt,tochild;
      Handle        global_handle,dechandle();
      int           **globals;

      if ( paramPtr->paramCount != 2 ) return;

      global_handle =
            (Handle) handlehex(*(paramPtr->params[0]));
      globals = (int **)*global_handle;
      tochild = (int) globals[TOCHILD];

      buf = *(paramPtr->params[1]);
      cnt = strlen(buf);
      cnt = auxwrite(tochild,buf,cnt);

      paramPtr->returnValue = dechandle(cnt,5);
}
```

This XFCN requires two parameters. The first is the handle that was returned by `forkpipexfcn`, and the second is a string that will be written to the child process. It is usually necessary to terminate a message to the child process with a newline character, and this can easily be done in HyperCard by concatenating the HyperCard constant `lineFeed` to the data string you are writing.

This XFCN returns the number of characters written to the child process. As usual, the return value is converted to a string and stored in a handle before being returned to HyperCard.

The XFCN `cleanupxfcn`

The last operation required is to close the open pipes, free any memory that the other XFCNs allocated along the way, and wait for the child process to terminate. It is *very* important that the child process be set to terminate before you call `cleanupxfcn`. This XFCN will not return to HyperCard until the child process has terminated, and the A/UX MultiFinder environment will be unavailable until it has. Here is a listing of the main code for this XFCN.

```
void do_xcmd(paramPtr)
XCmdPtr     paramPtr;
{
      long        status;
      Handle      global_handle,hexhandle();

      if ( paramPtr->paramCount != 1 ) return;
      global_handle = (Handle) handlehex(*(paramPtr->params[0]));
      status = auxcleanup_fork_pipe(global_handle);
      paramPtr->returnValue = hexhandle(status,8);
}
```

This XFCN requires one parameter, which is the handle that was returned by `forkpipexfcn`. It returns the exit status with which the child process terminated.

The HyperTalk scripts

With the building blocks provided by the four XFCNs just described, we can implement the UNIX Mail Reader application. The HyperCard stack for this application consists of two cards; the first card named, "headers," is used to display the list of UNIX mail messages, and the second card, named "message," is used to display each message. The application's behavior is specified in HyperTalk scripts associated with the two cards contained in the stack, and the buttons and fields contained on those cards.

The script for card "headers"

The first card in the UNIX Mail Reader stack displays the list of UNIX mail messages for the user who has opened the stack. There are two handlers in the script for this stack. The first is executed when the stack is opened, and the second is executed when the stack is closed. The content of these two handlers is shown in the following code:

```
on openStack
  global global_handle,linecount
```

```
  put empty into cd field one
  put empty into global_handle
  put forkpipexfcn("/usr/bin/mailx") into global_handle
  if global_handle is empty then go home
  put fgetsxfcn(global_handle,2500) into buf
  if word 1 of buf is "No" then
    put cleanupxfcn(global_handle) into status
    put empty into global_handle
    play "No mail"
    answer "Sorry, no mail"
    go to home
  else
    play "mail.sound"
    put fgetsxfcn(global_handle,2500) into buf
    repeat with linecount = 1 to 9999
      put fgetsxfcn(global_handle,250) into buf
      if length(buf) = 0 then exit repeat
      put buf into line linecount of cd field one
    end repeat
    subtract 1 from linecount
  end if
end openStack

on closeStack
  global global_handle
  if global_handle is not empty then
    answer "Update Message Queue?" with "Yes" or "No"
    if It is "No" then
      put writexfcn(global_handle,("x" & lineFeed)) into writecount
    else
      put writexfcn(global_handle,("q" & lineFeed)) into writecount
    end if
    put cleanupxfcn(global_handle) into status
    put empty into global_handle
  end if
  put empty into cd field one
  play "bye.sound"
end closeStack
```

The `openStack` handler initializes the fields and variables used by the stack. It then calls `forkpipexfcn` to execute the UNIX program `/usr/bin/mailx`. The value returned by `forkpipexfcn` is checked and then saved in a HyperCard variable called `global_handle`.

The handler then reads the first line of output produced by the `mailx` program using `fgetsxfcn`. The timeout value used is 2500, which is a large enough value to allow the child process to start up and to fill the pipe with the first line of output. If the first line of output begins with the word "No," there are no mail messages for this user, and the handler plays a recorded sound informing the user of this fact, then displays an alert box in confirmation.

If mail messages exist, the handler plays a recorded sound informing the user that he or she has mail. It then reads and ignores the next line of output, which contains the count of the messages. The script then loops while reading output from the child until no output is received. The timeout value used in this loop is 250, which is long enough to fill the pipe with the next line of output but not so long that a noticeable delay will result after the last line is read. Each line read is placed into the next line of the field named "one," which is on card "headers." After the last line has been read, the variable `linecount` is set to the number of mail messages available.

The `closeStack` handler checks `global_handle` to see if a child process exists. If so, it asks the user whether to update the message queue. If the user responds by clicking the Yes button, the handler exits the `mailx` program by sending the `q` command; otherwise it exits the `mailx` program by sending the `x` command.

The handler then calls `cleanupxfcn` to wrap up the session, cleans up some fields and variables in the stack, and plays a sound telling the user that it is done.

The script for field "one"

The field on the card "headers" that contains the list of messages has a script that handles reading a selected message. To select a message the user points to the desired message header in the list of messages and clicks the mouse button. That causes the following handler to be executed:

```
on mouseUp
  global global_handle,linecount,vline
  put (item 2 of the clickLoc) + (the scroll of cd field one) into
vline
  divide vline by the textHeight of cd field one
  put trunc(vline+.6) into vline
  if vline <= linecount then
    select line vline of cd field one
    if the textStyle of the selectedLine is italic then
      beep 1
    else
      put writexfcn(global_handle,(vline & lineFeed)) into
writecount
```

```
      play "ZoomUp"
      go to card 2
      put fgetfxfcn(global_handle,250) into cd field msgx
    end if
  else
    beep 1
  end if
  select empty
end mouseUp
```

This handler figures out which message header the user clicked (using a computation based on the click location and the text height). It checks to see that the user clicked a valid message by validating the message number and checking if the message header is italicized. (An italicized message header signifies a message that the user has read and deleted.)

After validating the message number, the handler writes to the `mailx` program telling it to deliver that message. The handler then uses `fgetfxfcn` to read the content of the requested message from the child process and place it into field "msgx" on the card named "message."

The script for button "Delete"

The card named "message" has a button named "Delete." If the user clicks this button the message being viewed will be marked for deletion. The following handler associated with this button implements this function.:

```
on mouseUp
  global global_handle,vline
  play "empty trash (flush)"
  put writexfcn(global_handle,("d" & lineFeed)) into writecount
  put empty into cd field msgx
  go to card 1
  select line vline of cd field one
  set textStyle of the selectedLine to italic
end mouseUp
```

The handler plays a humorous sound to indicate that a message is being deleted, sends the `d` command to the `mailx` program to mark the message for deletion, and changes the text style of the message header for this message to italic. This lets the handler for field "one" know that the message is no longer available for viewing.

The script for button "Return"

The card named "message" also contains a button named "Return," which is used to return to the card named "headers" to select another message or exit the stack. The handler for this button is shown below. It simply plays a sound, clears the field named "msgx" and returns to the card named "headers."

```
on mouseUp
  play "ZoomDown"
  put empty into cd field msgx
  go to card 1
end mouseUp
```

Summary

This sample application has shown in some detail the techniques that can be used to create Macintosh-style interfaces to UNIX commands. The creative developer will be able to take it and with minimal effort produce many more useful and sophisticated applications using HyperCard and the A/UX system call library.

Chapter 3 **Commando**

This chapter explains how you can write Commando dialog scripts to provide a Macintosh front-end for your UNIX® applications.

Commando lets you create CommandShell command lines by selecting controls within Macintosh dialog boxes. Controls direct the placement of options on the command line. By selecting a particular control, a specific option can be placed on the command line. The command lines thus constructed are placed in a CommandShell window for execution or optionally executed in a subshell.

This chapter begins with a discussion of dialog boxes in general; those familiar with this subject may want to turn directly to the section "Commando Dialog Boxes."

Introduction

The Macintosh computer provides you with visual cues when you communicate with an application, among them are the **controls** used in dialog boxes. Controls allow you to change the way an application functions; when a particular control is used it can place a specific option on the command line. The use of dialog boxes provides a consistency of interface across applications that decreases learning times for new applications and increases retention times for completed tasks. By implementing this interface on UNIX applications already running on A/UX, programmers and developers can increase the effectiveness of users working with the application.

Users who are relatively new to command-line interfaces often do not take the time necessary to learn all the intricacies needed to make full use of a program's features. Further, they are often frustrated in their attempts to use applications because it is not obvious what options are available, or what the application will do if they enter a given option. This is where Commando can help. Because Commando translates between visual controls and command-line options, users can see at a glance what an application can do and know what options are available. Further, Commando includes a context-sensitive help feature, so users receive an explanation of each control's effect as they click it. Programmers can save time because they will not have to explain the workings of the application time and time again.

Commando lets you create command lines using the controls within Macintosh dialog boxes. This makes invocation of even complex commands much easier, since users have feedback on what the command will do before they execute it. This also benefits occasional users of UNIX, it frees them from having to memorize the options or arguments associated with various commands. Even UNIX gurus appreciate this feature, since few have learned all the options of the more than 500 UNIX commands.

The contents of Commando dialog boxes are specified in *dialog scripts* written according to the Commando script language, which is discussed in detail later in this chapter. Much of the work of laying out the dialog boxes, including automatic vertical spacing, is done for you. This leaves you free to concentrate on the logical presentation order of the controls.

The steps you typically follow to create a Commando dialog are quite simple:

1. Copy a Commando dialog script from an existing command having similar controls. Scripts for all the Commando dialogs are kept in directories in `/mac/bin/cmdo`.
2. Modify the script to reflect the controls for your application or utility.
3. Test and debug the script.
4. Make sure the script has read-only permission.
5. Move the script to the appropriate directory in `/mac/bin/cmdo`.

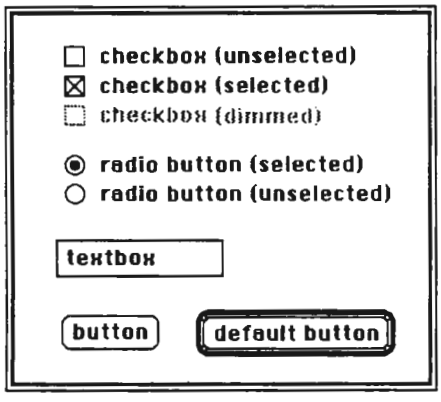
These steps are described in detail later in this chapter.

Macintosh dialog boxes

Dialog boxes provide the user with several visual cues (see Figure 3-1). The use of dialog boxes is governed by several conventions:

- Checkboxes allow users to select an option individually; these are the default controls in Commando.
- Radio buttons allow users to select mutually exclusive options.
- Text boxes allow (or require) users to enter additional information.
- Buttons allow users to select files or lead to further dialog boxes.
- Controls that cannot be selected are dimmed.

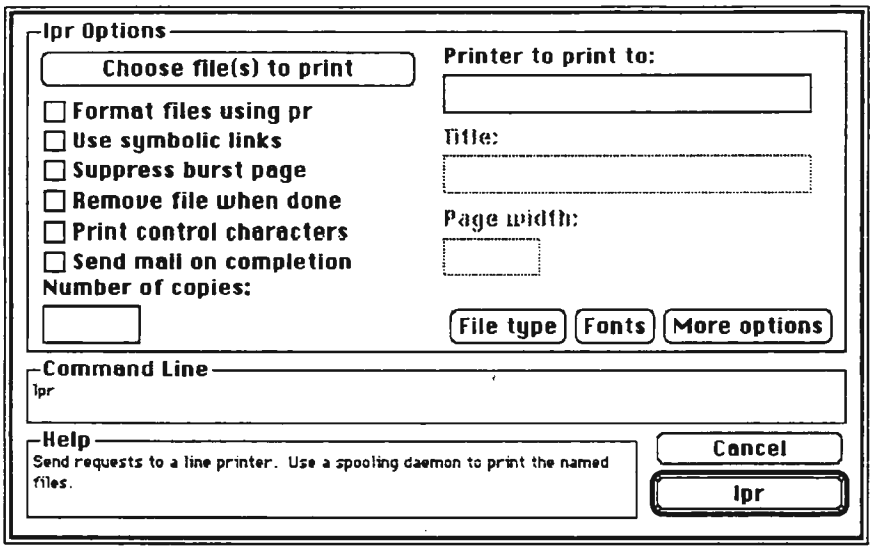
- **Figure 3-1** Schematic dialog box



Commando dialog boxes

All Commando dialog boxes have similar structures, though the controls for the command they represent will be different. Figure 3-2 shows a representative dialog box for a UNIX command. Each dialog shows the current command line being built, a box of Help information, and buttons to send or cancel the command. Each screen also has an area to select among the various options of the command. Each dialog box can have multiple controls, allowing command lines to be arbitrarily complex. Further, each command may have several nested dialog boxes; in Figure 3-2 the File type, Fonts, and More options buttons each lead to a nested subdialog.

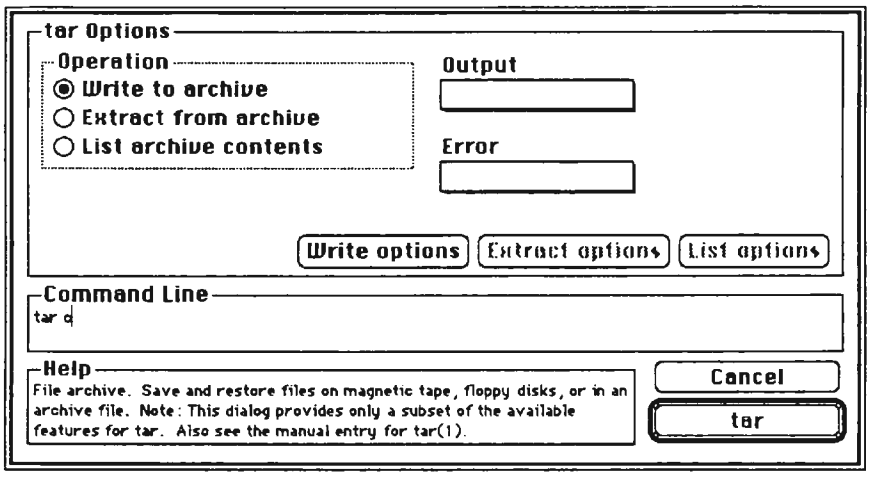
■ **Figure 3-2** Commando dialog box for the UNIX command `lpr`



Controls can be set up to enable other controls. In Figure 3-2, the title and page width controls are disabled because they are used only when the option "Format files using pr" is selected. Since this control hasn't been selected, the title and page width controls are inaccessible. Examples of this enabling feature are shown later in this chapter in the section "Commando Script Language."

Figure 3-3 shows another dialog box, this one for the `tar` command. The Operation controls, which control whether the program will read from or write to the backup media, are mutually exclusive and thus are implemented as radio buttons. The buttons giving access to dialog boxes containing further controls are enabled only when their corresponding radio button has been selected.

■ **Figure 3-3** Commando dialog box for the UNIX command `tar`



The Commando script language

The Commando script language helps you to create well-designed Commando dialog boxes quickly. Commando scripts allow users to start an application by double-clicking an icon or by invoking the application's dialog script from the command line. The resulting dialog boxes allow the user to select various flags and options, then pass the command line to the CommandShell for execution. By using dialog boxes developed through Commando, you can give your applications the front-end of a Macintosh application without changing the code of your UNIX application.

Dialog box layout

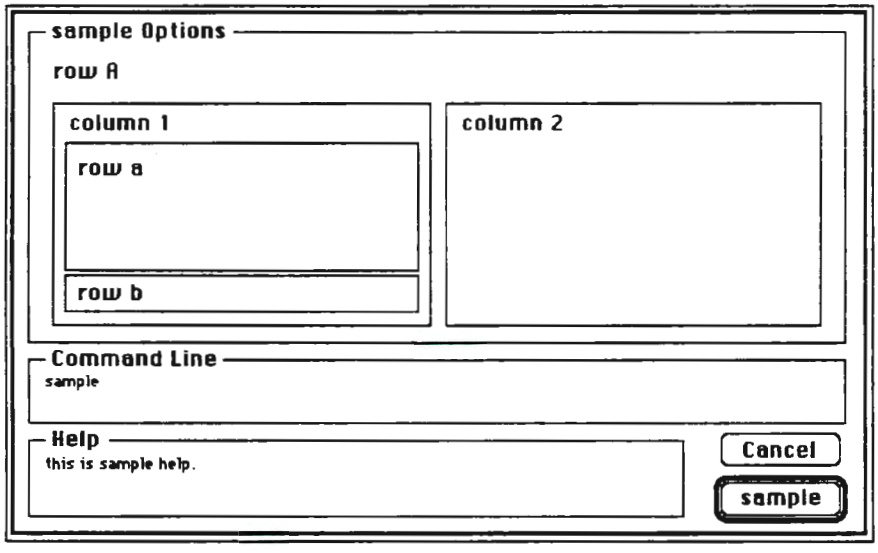
All Commando dialogs have several aspects of their layout in common: all will have labeled Options, Command Line, and Help boxes (see Figure 3-4). All will have button controls in the lower-right corner of the box allowing you to cancel the displayed dialog box, or (by default) complete your selections and send the command line to CommandShell.

The Option box of the dialog is laid out in rows and columns. There can be several rows within a given box. You can have multiple columns within a row, and multiple rows within a column.

Within a column or row are various command controls: buttons, checkboxes, text boxes, and radio buttons define how the command line will be built. Additional outline boxes can be added to group similar functions visually. Optional definitions may require or enable controls. Buttons leading to other dialog boxes can be included.

Figure 3-4 shows an Option box layout having two rows, a and b, enclosed within column 1, and the two columns, 1 and 2, enclosed within one large row, A. The various rows and columns are indicated by rectangles and names (in operation, Commando will not draw these rectangles or insert the names unless you specifically direct it to). In this simple layout example, no programmer-defined controls are shown.

■ **Figure 3-4** Dialog box layout example



Just as all Commando dialogs have some structures in common, all scripts have some structures in common. The beginning of the script always defines the name of the command, in this case "sample," by using the keyword `command name`. The name of the command appears in the default invocation button, in the Command Line box, and at the top left of the Option box (see Figure 3-4). Next, the keyword `help` defines the message shown in the Help box when you are not clicking a specific control. The maximum length of a help message varies with the length of the command name, but roughly 200 characters can be included.

The remainder of the script defines rows and columns of controls. Scripts reflect the structure displayed. If you want multiple columns within a row, column definitions are nested within the row definition. Each definition for a particular row or column is enclosed by braces. Row definitions begin with the keyword `row`, and column definitions with the keyword `column`. The braces may enclose other layout or control keywords, which will further affect the appearance of the dialog box. (Commando automatically adjusts the vertical size to include the defined controls.) Each keyword begins on its own line in a dialog script. (Complete listings of keywords are given in Tables 3-2 and 3-3.)

The script shown in Listing 3-1 reflects the structure displayed in Figure 3-4. Definitions for the innermost rows, a and b, are nested within column 1. The definitions for columns 1 and 2 are nested within row A. This simple example does not include any control specifiers; they would be enclosed by the braces between the beginning and end of the definition of a column or row.

- *Note:* Both within "real" dialog scripts and in the following examples comments are bracketed by slashes and asterisks: `/* this is a comment */`. Comments are used in the following examples to point out specific features of dialog scripts. Comments can be only one line long.

■ **Listing 3-1** Dialog box layout example script

```
command name "sample"
help "this is sample help."
row {                   /* this begins row A */
  column {                 /* this begins column 1 */
         row {                /* this begins row a */
             }                /* this ends row a */
         row {                /* this begins row b */
             }                /* this ends row b */
           }                  /* this ends column 1 */
  column {                    /* this begins column 2 */
           }                  /* this ends column 2 */
    }                     /* this ends row A */
```

The example script in Listing 3-1 shows how the layout is theoretically arranged on the screen. The following sections examine more realistic examples.

Layout examples

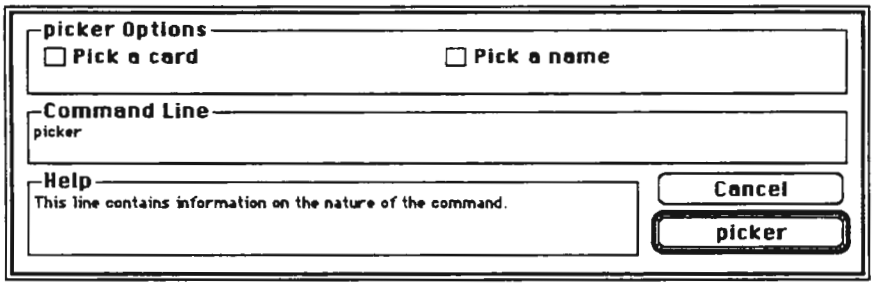
This section contains examples of the layout of controls. The controls themselves are not discussed in depth; they are discussed later in the chapter in the section "Control Examples."

Row example

Figure 3-5 shows a trivial example containing only two programmer-defined controls. In this example they are both checkboxes.

◆ *Note:* In the following examples a sample Commando dialog box is shown first, and the dialog script that produced it is shown next.

■ **Figure 3-5** Single row dialog box



The dialog script that produced this dialog is shown in Listing 3-2.

■ **Listing 3-2** Single row dialog script

```
command name "picker"                          /* command name */
help  "This line contains information "        /* help message */
      "on the nature of the command."
row {                                          /* begin only row */
    option name "Pick a card"                  /* first control */
        prefix "-p"
        help "[-p] randomly pick a card."
    option name "Pick a name"                  /* second control */
        prefix "-n"
        help "[-n] randomly pick a name."
    }                                          /* end of row */
```

A line-by-line dissection of this script shows several general points worth noting:

- The first line,

  ```
  command name "picker"
  ```

  defines the command name. As mentioned previously, it is automatically put on the command line being built (in the Command Line box of Figure 3-5), in the command button (at the lower left of Figure 3-5), and in the Options label at the top of the dialog box.

- The second and third lines,

  ```
  help  "This line contains information "
        "on the nature of the command."
  ```

  define the help message for the command displayed in the bottom Help box. The message can span several lines, which are concatenated when the dialog is constructed. The help message for the command is displayed whenever the mouse button is up.

- The fourth line,

  ```
  row {
  ```

  specifies construction of a row. Controls between this point and the closing brace (on the last line) will all be placed on the same row.

- On the fifth line,

  ```
  option name "Pick a card"
  ```

  the `option name` for the first control defines how the control will be labeled.

- On the sixth line,

  ```
  prefix "-p"
  ```

  the `prefix` line defines what characters are placed on the command line being built when this control is selected.

- On the seventh line,

  ```
  help "[-p] randomly pick a card."
  ```

  the `help` line following an `option name` keyword defines what will appear in the Help box when the pointer is positioned over this control and the mouse button is down.

- The eighth through tenth lines,

  ```
  option name "Pick a name"
     prefix "-n"
     help "[-n] randomly pick a name."
  ```

  specify another control; each control consists of at least the option name, prefix, and a help message.

- The eleventh line,

  ```
  }
  ```

has the closing brace for the first row.

Commando automatically divides a row into columns in order to space the controls. In Figure 3-7 there are two controls, so two columns are used for spacing. This spacing can affect the length you choose for control names.

Multiple row example

Figure 3-6 shows a dialog box with different rows having different numbers of controls. The first row contains the three controls: "Pick a card," "Pick a name," and "Pick a spot." The second row contains the two pop-up menus Output and Error.

This example shows what the dialog box looks like if the pointer is positioned on the "Pick a card" option and the mouse button is down. The control shows that it is selected (there is an x in the checkbox), the `-p` prefix shows in the Command Line box, and the help box displays the message associated with that option. When the mouse button is released, the control remains selected and the prefix remains in the command line being built, but the help message will revert to the message for the command itself.

- **Figure 3-6** Multiple row dialog box

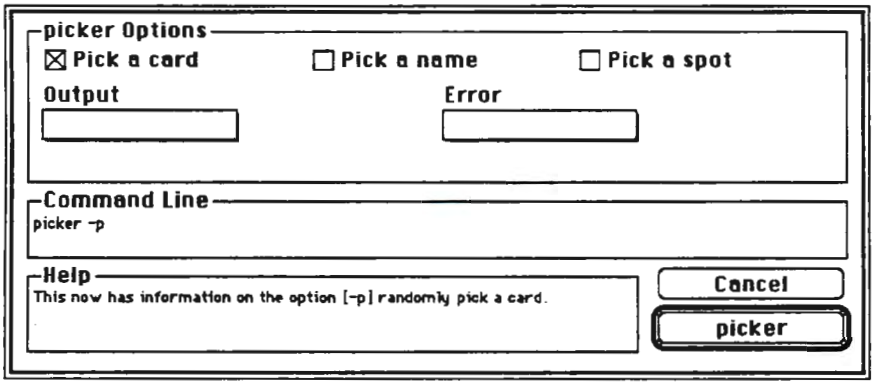


Figure 3-6 and Listing 3-3 show a new point: the type of control displayed in the dialog changes when the keyword within an `option name` section is changed. Note that the options within the shaded area of Listing 3-3 use the keywords `outpopup` and `errpopup`. These keywords create different kinds of controls than those created by the default checkbox. The various types of controls are covered in depth in the section "Control Examples." These examples again demonstrate the automatic building of columns within rows. The first row has three controls and is displayed in three columns, while the second row has two controls and is displayed in two columns. The vertical spacing is again adjusted automatically to allow room for the controls.

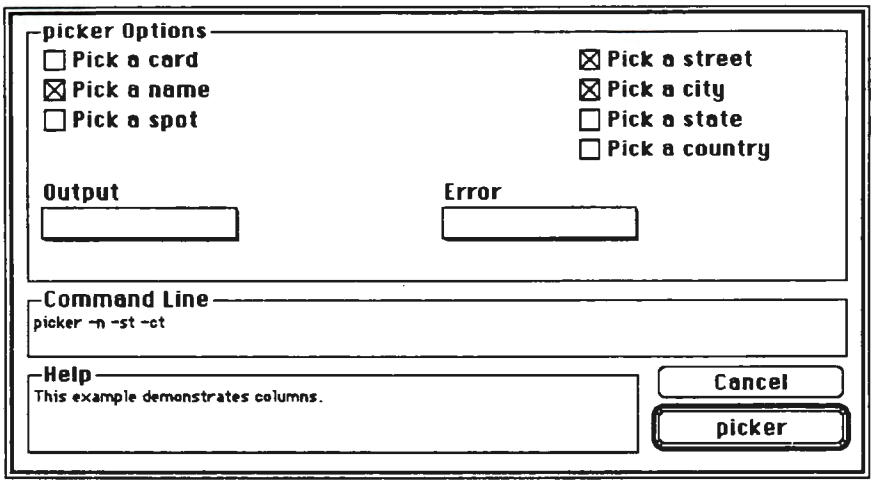
■ **Listing 3-3** Multiple row dialog script

```
command name "picker"                          /* command name */
help  "This line contains information "        /* help message */
      "on the nature of the command."
row {                                       /* start first row */
    option name "Pick a card"                   /* first control */
        prefix "-p"
        help "This now has information on the option "
             "[-p] randomly pick a card."
    option name "Pick a name"                   /* second control */
        prefix "-n"
        help ""This now has information on the option "
             "[-n] randomly pick a name."
    option name "Pick a spot"                   /* third control */
        prefix "-s"
        help ""This now has information on the option "
             "[-s] randomly pick a spot."
    }                                       /* end first row */
row {                                       /* start second row */
    option name "output"                        /* first control */
        outpopup
    option name "error"                         /* second control */
        errpopup
    }                                       /* end second row */
```

Column example

The following examples (Figure 3-7 and Listing 3-4) demonstrate the explicit definition of a column. Multiple columns can be defined within a row, with the horizontal spacing divided equally by the defined number of columns. Multiple columns containing different numbers of controls can be contained within the same row. Commando automatically adjusts the vertical height of the dialog box based on the number of controls in a particular column (within limits, of course).

■ **Figure 3-7** Multiple column dialog box



In the upper shaded area of Listing 3-4 the keyword `column` is used within the first row to put all three controls in the same column. The unshaded area between the shaded areas contains a dummy column, one with nothing between its braces; it is used to create the blank column. The lower shaded area starts another column specification, this time putting four controls in the column. Commando again takes care of adjusting the dialog box's vertical spacing.

■ **Listing 3-4** Multiple column dialog script

```
command name "picker"                              /* command name */
help  "This example demonstrates columns."         /* help message */

row {                                        /* start first row */
    column {                                   /* start first column */
        option name "Pick a card"                  /* first control */
            prefix "-p"
            help "[-p] randomly pick a card."
        option name "Pick a name"                  /* second control */
            prefix "-n"
            help "[-n] randomly pick a name."
        option name "Pick a spot"                  /* third control */
            prefix "-s"
            help "[-s] randomly pick a spot."
        }                                        /* end first column */
    column {}                  /* dummy second column for spacing */
```

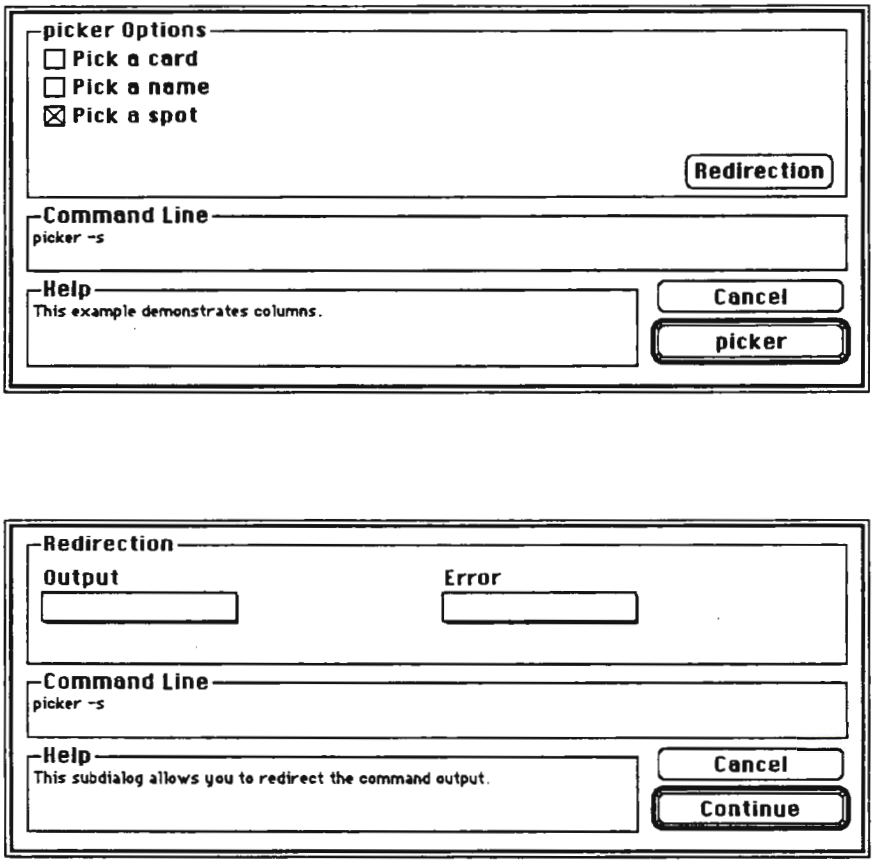
```
    column {                                    /* start third column */
        option name "Pick a street"                 /* first control */
            prefix "-st"
            help "[-s] randomly pick a street."
        option name "Pick a city"                   /* second control */
            prefix "-ct"
            help "[-n] randomly pick a city."
        option name "Pick a state"                  /* third control */
            prefix "-sta"
            help "[-n] randomly pick a state."
        option name "Pick a country"                /* fourth control */
            prefix "-c"
            help "[-s] randomly pick a country."
        }                                           /* end third column */
    }                                           /* end first row */
row {                                           /* start second row */
    option name "output"                            /* first control */
        outpopup
    option name "error"                             /* second control */
        errpopup
    }                                           /* end second row */
```

Nested dialog box example

Figure 3-8 shows the next step in changing the structure, the addition of a button leading to a further dialog box (nested dialog boxes are also referred to as *subdialogs*). Here a new dialog named "Redirection" was added to the first dialog box. Clicking the Redirection button leads to the subdialog, shown as the second dialog of Figure 3-8.

Note that the second dialog box (the lower box shown in Figure 3-8) has a Continue button that returns you to the first dialog box. Multiple nested dialog boxes can be specified (see Figure 3-2, "Commando Dialog Box for the UNIX Command `lpr`").

■ **Figure 3-8** Further dialog example



The shaded area of Listing 3-5 shows the addition of a button leading to a further dialog box. Here a new dialog named "Redirection" was added to the original dialog box. Note that the button to access this further dialog was automatically sized to hold the name.

■ **Listing 3-5** Further dialog script

```
command name "picker"                          /* command name */
help  "This example demonstrates columns."  /* help message */

row {                                        /* start first row */
    column {                                    /* start first column */
        option name "Pick a card"
            prefix "-p"
            help "[-p] randomly pick a card."
        option name "Pick a name"
            prefix "-n"
            help "[-n] randomly pick a name."
        option name "Pick a spot"
            prefix "-s"
            help "[-s] randomly pick a spot."
           }                                        /* end first column */
    }                                          /* end first row */
dialog name "redirection"                      /* start second dialog */
help  "This subdialog allows you to "       /* help message */
      "redirect the command output."
row {                                          /* start first row */
    option name "output"
        outpopup
    option name "error"
        errpopup
    }                                          /* end first row */
```

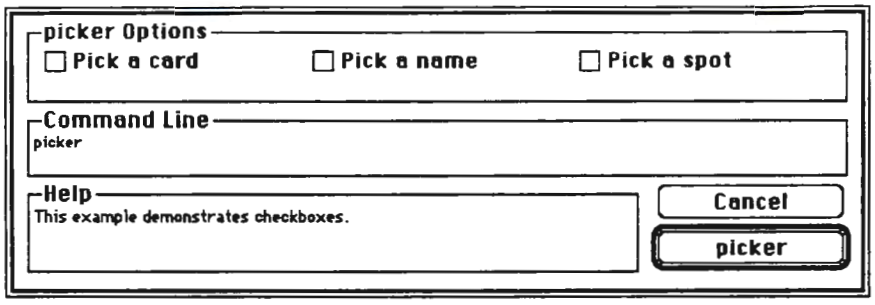
Control examples

Places in a dialog where the user can make a choice are called *controls*. These include checkboxes, radio buttons, text boxes, and buttons. Keyword specifiers define all controls available from the dialog box. The type of control is usually specified (a checkbox is the default). In addition, enabling and requirement dependencies can be defined (see the section "Dependencies" below). The various types of controls are discussed in the following sections.

Checkbox

This is the default control type, a square box that the user either selects or deselects by clicking it. The user selects each checkbox individually. Figure 3-9 shows examples of this type of control.

■ **Figure 3-9** Checkbox example dialog



Listing 3-6 shows the script that produced the Checkbox example dialog; the shaded area contains a representative checkbox definition. You define the `option name`, `prefix`, and `help` specifiers for every checkbox. Place the text following each of these keywords between double quotation marks. The maximum number of checkboxes in a column is ten.

■ **Listing 3-6** Checkbox example script

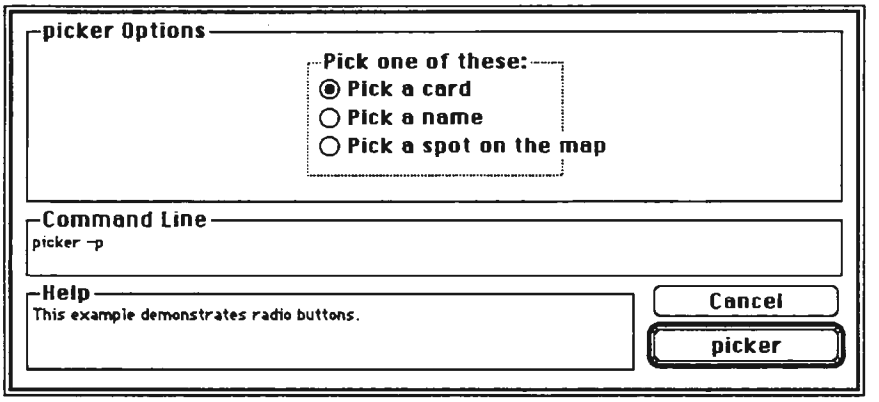
```
command name "picker"
help "This example demonstrates "
     "checkboxes."
row {
    option name "Pick a card"
        prefix "-p"
        help "[-p] randomly pick a card."
    option name "Pick a name"
        prefix "-n"
        help "[-n] randomly pick a name."
    option name "Pick a spot"
        prefix "-s"
        help "[-s] randomly pick a spot."
    }
```

Radio buttons

Radio buttons are similar to checkboxes but provide users with mutually exclusive controls; an example is shown in Figure 3-10. Users see associated radio buttons aligned in columns; a box, referred to as a *named box*, usually surrounds radio buttons to visually indicate that they are related. Commando automatically selects the first radio button in a set for the user.

Due to an intentional programming error, one of the labels is too long, and has extended outside the named box. This demonstrates that you must choose control labels that will fit in the column they are in.

■ **Figure 3-10** Radio button example dialog



The shaded area of Listing 3-7 shows a definition for a set of radio buttons. The definition starts with the keyword `radio buttons` and encloses the set of individual controls in braces. Specify the keywords `option name`, `prefix`, and `help` for each button. Use a box to visually indicate the grouping of the radio buttons. To do this, use the keyword `name` (for a named box, as shown in Listing 3-7) or the keyword `box` (for an unnamed box) within the radio button definition. Commando automatically aligns radio buttons into columns. A maximum of seven radio buttons can be grouped in a set. By default, Commando selects the first radio button in a set, so make the first control the one that the user will most often choose.

Figures 3-16 shows that one of the labels is too long. You can avoid this by using concise labels or wider columns.

■ **Listing 3-7** Radio button example script

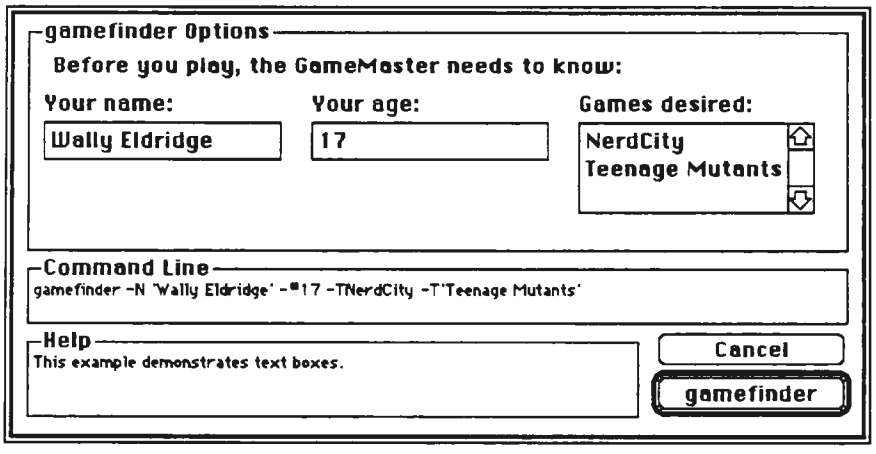
```
command name "picker"
help "This example demonstrates "
     "radio buttons."
row {
    column {}                          /* dummy column for spacing */
```

```
    radio buttons {              /* specify a set of radio buttons */
    name "Pick one of these"     /* create a named grouping box */
        option name "Pick a card"
            prefix "-p"
            help "[-p] randomly pick a card."
        option name "Pick a name"
            prefix "-n"
            help "[-n] randomly pick a name."
        option name "Pick a spot on the map"
            prefix "-s"
            help "[-s] randomly pick a spot."
              }
    column {}                          /* dummy column for spacing */
    }
```

Text boxes

A text box allows the user to enter text to be used in the control string. Text boxes are the width of the current column. Figure 3-11 shows an example of the use of these text input types. Note that when an input string contains blanks, Commando automatically encloses the string in single quotation marks to avoid confusing the shell. (For example, see the name Wally Eldridge shown in the Command Line box.)

- **Figure 3-11** Text box example dialog



Listing 3-8 shows the script used to create the dialog in Figure 3-11. As with a checkbox, specify the keywords `option name`, `prefix`, and `help` for each text box. Use one of the keywords `string` or `stringlist` to indicate the type of data to be input by the user. The keyword `string` allows entry of a single line of text, while `stringlist` allows entry of several lines, each of which is prefaced on the command line with the defined `prefix`. Text boxes are the width of the current column. You can put a maximum of three `string` controls in a column. You can put a maximum of two `stringlist` controls in a column.

As mentioned previously, when an input string contains blanks Commando automatically encloses the string in single quotation marks to avoid confusing the shell. Putting the keyword `dontquote` on the next line after the keyword `command name` will turn off this quoting feature for the entire dialog (this variant is not shown).

- *Note:* Some UNIX commands insist that no spaces come between an option and its argument on a command line. In these cases, include control characters in the prefix definition to remove the spaces normally inserted. This is indicated in the text by a circumflex (^) before a character. For example, `^Y` indicates CONTROL-Y, and is placed just after an option to remove the space between the option and its argument.

△ **Important** Control characters do not normally print well; consequently printouts of your dialog scripts may not show all the characters that are actually there. A circumflex followed by a letter is *not* a substitute for a control character. △

Listing 3-8 shows various ways that text input is translated to the command line. The code in the top shaded area of the figure formats input text on the command line with a space between the input text and the prefix. The code in the unshaded area between the shaded areas defines a control having no space between the prefix and the text because of the CONTROL-Y at the end of the prefix. You can also remove the space before a prefix by using a CONTROL-H before the letter of the option. Each of these control characters can be used only once per option, though both can be used on a single option. The code in the bottom shaded area of the figure shows how you can put several arguments having the same prefix on the command line, using the keyword `stringlist` rather than `string`.

■ **Listing 3-8** Text box example script

```
command name "gamefinder"
help "This example demonstrates text boxes."
row {
    option name "Before you play, the GameMaster needs to know:"
        text                                          /* first control */
    }
row {}                                        /* dummy row for spacing */
row {
    option name "Your name:"                          /* second control */
        prefix "-N"
        help "[-N] This enters your name."
        string
    option name "Your age:"                           /* third control */
        prefix "-#^Y"      /* use Control-Y for spacing */
        help "[-#]  Your age determines the play level."
        string
    option name "Games desired:"                      /* fourth control */
        prefix "-T^Y"      /* use Control-Y for spacing */
        help "[-T] Specify all the games you want to try."
        stringlist
    }
```

Text

Listing 3-8 also shows the use of the keyword `text` on the unshaded line labeled "first control." This control does not allow input, but simply places text in the dialog. Unlike controls that allow input, you don't specify the keywords `prefix` or `help`.

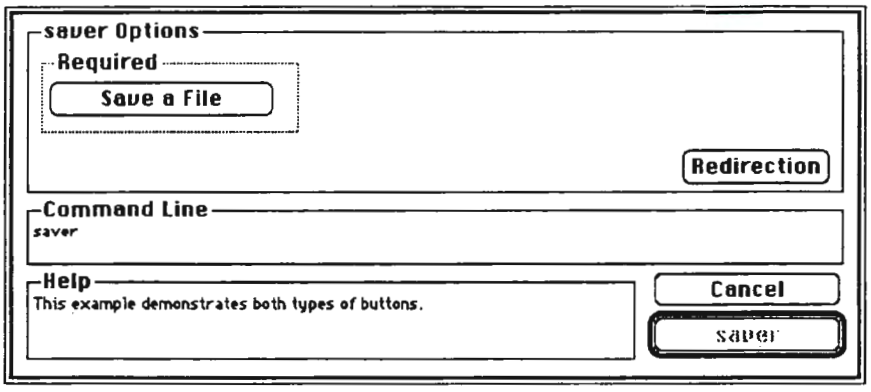
Buttons

With dialog buttons the user can

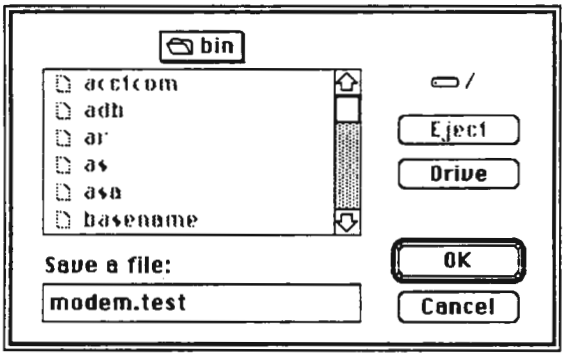
- open further windows that allow access to files on which to operate and directories in which to save files
- open a nested dialog box, allowing choices of additional options

Dialog buttons are different from radio buttons, which select between mutually exclusive actions. Because they have a different function, they are a different shape. Figure 3-12 shows examples of both types of dialog buttons; their names indicate their purpose. If the user clicks the "Save a file" button a second dialog box appears (see Figure 3-13) and shows the standard Macintosh file dialog box for selecting a new filename. After the user selects a file, the original dialog box reappears (see Figure 3-14; note the filename in the Command Line box). If the user clicks the "Redirection" button, the dialog shown in Figure 3-15 comes up, allowing a choice of redirection options.

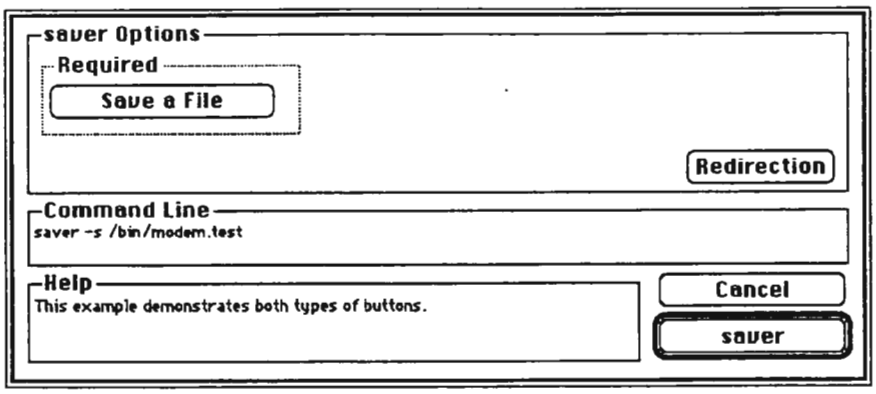
■ **Figure 3-12** Button example: Initial dialog box



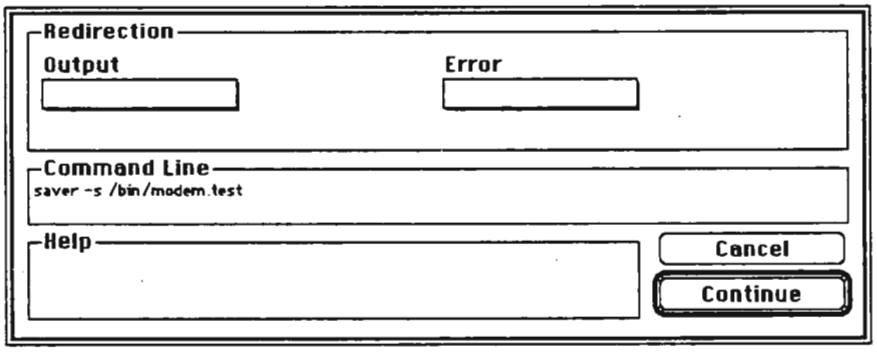
■ **Figure 3-13** Button example: Save a file dialog box



■ **Figure 3-14** Button example: Save a file control was selected



■ **Figure 3-15** Button example: Redirection subdialog box



With dialog buttons you can call file dialogs or call a subdialog. With dialog buttons you don't have to use the keyword `prefix`, but it is good practice to always use the keyword `help` with them, though it is not required. You create file dialog buttons by putting one of the following keywords after the keyword `option name`:

```
file filelist newfile directory dirlist dirsandfiles filesanddirs
```

The purpose of each keyword is listed in Table 3-1. (A complete listing of keywords can be found in Tables 3-2 and 3-3.)

■ **Table 3-1** File dialog keywords

Keyword	Description
`file`	Presents the single file choice menu.
`filelist`	Presents the file list choice menu.
`newfile`	Presents the new file creation menu.
`directory`	Presents the single directory choice menu.
`dirlist`	Presents the directory list choice menu.
`dirsandfiles`	Presents the file/directory choice menu. Same as `filesanddirs`.
`filesanddirs`	Presents the file/directory choice menu. Same as `dirsandfiles`.

To redirect either the standard or error output, use the keywords `outpopup` and `errpopup`. You can use `outpopup` alone; however, to use `errpopup`, you must also use `outpopup`.

To create dialog buttons that open a subdialog box, use the keyword `dialog name`. You must place this keyword after the close of a row definition (as is shown in the lower shaded area of Listing 3-9). Define the name of the button with a text string (between double quotation marks) following the keyword. You can put a maximum of six buttons in a column.

Listing 3-9 shows the script that produced the dialogs in Figures 3-12 through 3-15. The first button in the script (in the upper shaded area of the figure) calls a file dialog, in this case to create a new file. The first button control is followed by two dummy columns to ensure that the button does not extend the entire width of the dialog. The second button (in the lower shaded area of the figure) opens a subdialog whose only components are the redirection pop-up menus.

These figures also illustrate the effects of a new keyword, `required`, found within the first control. The keyword `required` has the effect of disabling the command button until a file has been selected (note the difference in the appearance of the button named "saver" between Figures 3-20 and 3-22). The keyword `required` can only be used on the first dialog of a script. The keyword `name` is used to place a box around the required control to notify the user to complete this control; this keyword has been seen before. These keywords are discussed further in the section "Dependencies."

■ **Listing 3-9** Button Example script

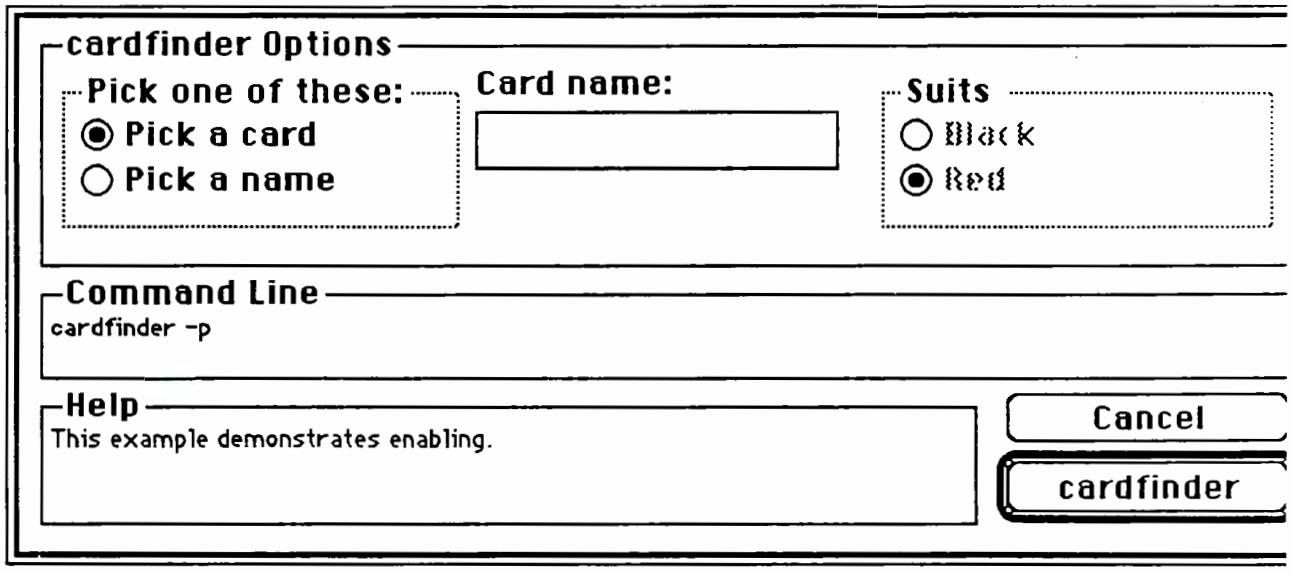
```
command name "saver"
help  "This example demonstrates both types of buttons."
row {
    column {
        name "Required"              /* let user know about this */
        option name "Save a file:"          /* first button */
            prefix "-s"
            help "[-s] Saves to chosen name."
            newfile                  /* get a new file */
            required              /* HAVE to get a new file */
            }
    column {}                    /* dummy column for spacing */
    column {}                    /* dummy column for spacing */
    }
dialog name "Redirection"                    /* second button */
row {
    option name "output"
        outpopup
    option name "error"
        errpopup
    }
```

Dependencies

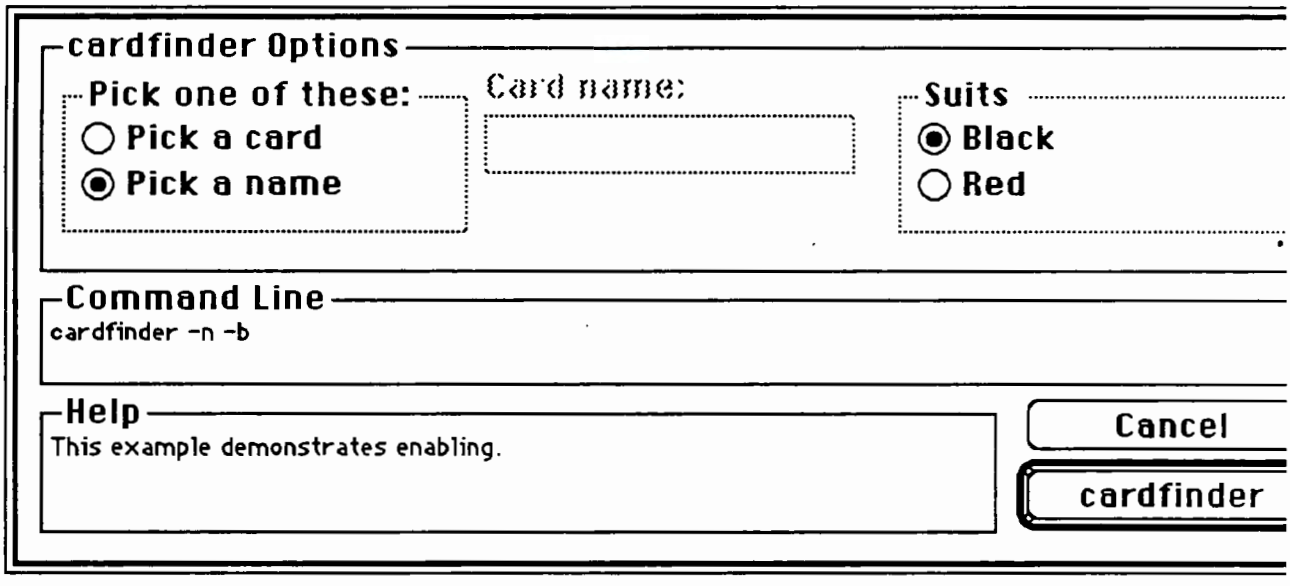
Controls can be selectively enabled, depending on the selection state of some other control. Controls that are disabled appear in gray type (and are said to be *dimmed*); once the enabling dependency is satisfied the control appears in black type. Users also can be required to select an option.

Figure 3-16 and Figure 3-17 show a control dependency example. In Figure 3-16, the "Pick a card" control is selected (it is the default) so the "Card name" control is enabled, while the "Suit" controls are disabled. In Figure 3-17, this order is reversed; the "Suit "control is now enabled, while the "Card name" control is disabled.

■ **Figure 3-16** Dependencies example: First control selected



■ **Figure 3-17** Dependencies example: Second control selected



Controls without dependencies are enabled by default; controls with dependencies are disabled by default. To disable a control, simply make its enabling dependent on another control by using the keyword `enables`.

You can enable controls in two ways:

- Specify the prefix that must be in effect (showing in the Command Line box). The prefix must be identical to that used in the control specification, including any control characters within the prefix.
- Specify the control's option name (the quoted text following keywords `option name` or `name`). For example, radio buttons are enabled as a set by enclosing them in a named box and putting the box's name in double quotation marks after the keyword `enables`.

In order for enabling dependencies to work, all dependent controls must be in the same dialog box. If necessary, place dependent controls together in a subdialog and enable a button that allows the user access to that subdialog.

You can require that users select an option by using the keyword `required`. The keyword `required` can be used only on the first dialog of a script. It is helpful to the user to enclose any required controls in a box named "Required" (see the examples in Figures 3-20 through 3-24).

Listing 3-10 shows the script used to created the dialogs shown in Figures 3-16 and 3-17. The first control enables the third control (see the first and third shaded areas of the figure), while the second enables the grouped fourth and fifth controls (see the second and fourth shaded areas of the figure). The first control enables a single control by prefix; you can use this method for all kinds of controls. The second control enables the grouped controls by name. Use this method for specifying a set of radio buttons, for individual buttons, and for controls with blank prefixes.

■ **Listing 3-10** Dependencies example script

```
command name "cardfinder"
help "This example demonstrates enabling."
row {
    column {
    name "Pick one of these:"
    radio buttons {
        option name "Pick a card"                    /* first control */
            prefix "-p"
            help "[-p] This allows selection of a card."
            enables "-c^Y"                  /* enable a single control */
        option name "Pick a name"                   /* second control */
            prefix "-n"
            help "[-n] Select a name."
            enables "Suits"          /* enable a group of controls */
                  }
           }
    option name "Card name:"                         /* third control */
        prefix "-c^Y"
        help "[-c] This enters the card name."
        string
    radio buttons {                  /* set up a group of controls */
    name "Suits"
        option name "Black:"                         /* fourth control */
            prefix "-b"
            help "[-b]  Select from black suits."
        option name "Red:"                           /* fifth control */
            prefix "-r"
            help "[-r]  Select from red suits."
                  }
    }
```

The order that options appear on the command line can be specified, in reverse order, by using the keywords `last1`, `last2`, and so on. An option with the keyword `last1` will appear last on the command line. An option with the keyword `last2` will appear next to last, and so on. This feature can be used within a dialog box, and is nested across dialog boxes. For example, the `last1` specification of an option in the first dialog box will be put on the command line after an option with the `last1` specification in any subdialog boxes.

Boxes

Outline boxes can be defined by using the keywords `box` or `name`. Each draws a box around a control or group of controls; the keyword `name` inserts a name at the top left of the box to identify its contents. The name can be as long as you like. However, if it is longer than the box it will overwrite the next column. Named boxes can be used to enable a group of radio buttons (see Figures 3-16 and 3-17, and Listing 3-10) The width of the boxes is the same as that of the current column. You can often make a dialog look better by inserting blank columns to reduce the width of the boxes (see Figures 3-7 and 3-10).

Leniencies

Commando is fairly forgiving when it comes to specifying column definitions. It is good about automatically creating columns, and usually the first column specification in a multiple column set does not need to be explicit. Radio buttons are automatically put into their own column. Commando is also reasonably well behaved as long as you don't try to put more than seven controls in a column (explicit or implicit).

Keywords

The following tables list the keywords used in Commando presented in two ways. The first is grouped by function, the second is a alphabetic listing.

- **Table 3-2** Commando keyword reference: by function

Keyword	Description
`command name "`*`name`*`"`	Sets the name of the command in the invocation button.
`help "`*`help string`*`"`	Sets the help message for this section.
`row { }`	Contains the contents of a row.
`column { }`	Contains the contents of a column.
`option name "`*`name`*`"`	Sets the name of checkboxes and/or buttons. Required for each control.
`prefix "`*`prefix string`*`"`	Adds *prefix string* to the command line.
`help "`*`help string`*`"`	Sets the help message for this section.
`radio buttons { }`	Defines a set of radio buttons. The braces enclose the set of controls.

`dialog name "`*name*`"`	Sets the name for a nested dialog box and the button to access it.
`outpopup`	Presents the standard output redirection menu. Required if `errpopup` is used.
`errpopup`	Presents the standard error redirection menu.
`dirsandfiles`	Presents the file/directory choice menu. Same as `filesanddirs`.
`directory`	Presents the single directory choice menu.
`dirlist`	Presents the directory list choice menu.
`filelist`	Presents the file list choice menu.
`file`	Presents the single file choice menu.
`filesanddirs`	Presents the file/directory choice menu. Same as `dirsandfiles`.
`newfile`	Presents the new file creation menu.
`required`	One of the controls referenced by this keyword must be selected.
`enables "`*specifier*`"`	Enables other controls to be used. The control to be enabled is specified by its prefix.
`disabled`	Obsolete keyword.
`dontquote`	Turns off the quoting mechanism for text input. Affects all text fields in a dialog script.
`string`	Allows string input. The input box string width is the width of the current column.
`stringlist`	Allows several string inputs. The input box string width is the width of the current column.

■ **Table 3-2** Commando keyword reference: by function

Keyword	Description
`text`	Displays the control name as text.
`number`	Obsolete keyword.
`last1..n`	Specifies the order of options. `last1` indicates the last option on the command line. `last2` is the next-to-last, and so on.
`box`	Puts an outline box around a control or group of controls.
`name`	Puts a named outline box around a control or group of controls.

■ **Table 3-3** Commando keyword reference: alphabetic

Keyword	Description
`box`	Puts an outline box around a control or group of controls.
`column { }`	Contains the contents of a column.
`command name "`*`name`*`"`	Sets the name of the command in the invocation button
`dialog name "`*`name`*`"`	Sets the name for a nested dialog box and the button to access it.
`directory`	Presents the single directory choice menu.
`dirlist`	Presents the directory list choice menu.
`dirsandfiles`	Presents the file/directory choice menu. Same as `filesanddirs`.
`disabled`	Obsolete keyword.
`dontquote`	Turns off the quoting mechanism for text input. Affects all text fields in a dialog script.
`enables "`*`specifier`*`"`	Enables other controls to be used. The control to be enabled is specified by its prefix.
`errpopup`	Presents the standard error redirection menu.
`file`	Presents the single file choice menu.
`filelist`	Presents the file list choice menu.
`filesanddirs`	Presents the file/directory choice menu. Same as `dirsandfiles`.
`help "`*`help string`*`"`	Sets the help message for this section.
`last1..n`	Used to specify the order of controls. `last1` indicates the last option on the command line. `last2` is the next-to-last, and so on.
`name`	puts a named outline box around a control or group of controls.
`newfile`	Presents the new file creation menu.
`number`	Obsolete keyword.
`option name "`*`name`*`"`	Sets the name of checkboxes and/or buttons. Required for each control.
`outpopup`	Presents the standard output redirection menu. Required if `errpopup` is used.
`prefix "`*`prefix string`*`"`	Adds *prefix string* to the command line.
`radio buttons { }`	Defines a set of radio buttons. The braces enclose the set of controls.
`required`	One of the controls referenced by this keyword must be selected.
`row { }`	Contains the contents of a row.
`string`	Allows string input. The input box string width is the width of the current column.
`text`	Displays the control name as text.

■ **Table 3-3** Commando Keyword Reference: alphabetic

Keyword	Description
`stringlist`	Allows several string inputs. The input box string width is the width of the current column.

Creating Commando dialogs

Creating new Commando dialogs is a three-step process. First, you write a new script. This usually involves copying a script that has controls similar to the ones you want to use, then modifying it to fit your application. Second, you test, and if necessary debug, the script. Third, you make the script read-only and move it to one or more places so it can be invoked by all the users on the system. Optionally you can compile the script into a resource.

As an introduction to the process of creating dialogs, the following section examines how dialogs are invoked.

Invoking Commando Dialogs

To invoke Commando from the CommandShell, you can use two methods. Enter

`cmdo` *commandname*

on the command line, or type

commandname

on the command line and choose Commando from the Edit menu (the keyboard shortcut for this method is *commandname* COMMAND-K).

When Commando starts, it first searches the path listed in the variable `$CMDODIR` for resources, then for dialog scripts. After that, Commando searches for resources, then dialog scripts, in the directory in `/mac/lib/cmdo` having the same first letter as the command name you are invoking. Finally, Commando searches your `$PATH` variable for resources (this may result in a long search if `$PATH` includes many directories).

Make sure that the commands on the command line created by your dialog script are locatable by the shell. The normal command search path is contained in the `$PATH` shell variable. By default this variable is set to `/bin:/usr/bin:/usr/ucb:/mac/bin:`, though this may be changed by system initialization files (such as `.profile` or `.login`).

Commando is also invoked when you double-click a UNIX application, utility, or shell script icon. This method is not efficient when you are testing dialog scripts.

Writing Commando dialogs

Although the Commando script language is reasonably straightforward, it is not foolproof. The Commando scripts that reside on each A/UX system (in `/mac/lib/cmdo/*/*`) have all been debugged and tested. Consequently, you can save time if you simply modify a script that already exists instead of trying to write your own script from scratch. This is especially true because some scripts use nonprinting control characters to enable controls, and such scripts are sometimes difficult to debug from printouts.

Testing Commando dialogs

Commando dialogs are easy to test, even when the script file is still open. When Commando is searching for script files, it searches the directories listed in the section "Invoking Commando Dialogs." Therefore, once you have written your script, simply place it in the directory within `/mac/lib/cmdo` that has the same first letter as your script's name. The file should have read permission for your users. If you've modified a file that already existed, you probably won't need to change the permissions. To set the permissions so the file is readable by everyone, use the command line

`chmod 444` *scriptname*

If you are using TextEditor to edit a Commando file, simply save the file (you don't have to close it) in the appropriate directory within `/mac/lib/cmdo`. Commando will interpret and run the last saved version of your script. If it doesn't perform or look quite right, simply edit the file, save it again, and reinvoke the script using one of the command lines discussed earlier in this chapter.

Compiling Commando dialogs

Compiling a script into a resource file allows you to customize its appearance. Various attributes, such as the size of dialog boxes and the shape of controls, can be modified using the Commando resource editor available in MPW.

To create a Commando resource use the command line

`cmdo` *scriptname* `-r -n -o` *outputfile*

This creates a resource file with the name *outputfile*. Move the file into a directory common to user's `$PATH`s, such as `/usr/bin`, so all users can access it. After the file is moved, it must be renamed to *scriptname* so Commando can locate the source dialog.

The command, the script, and the compiled resource must all have the same name (the resource file has a leading `%`).

Dialog design guidelines

This section offers general guidelines to assist you in planning your Commando dialogs. It is not meant to be authoritative, but does present what has been found to work best. If the needs of your applications demand it, you are free to do anything you want, but keep in mind that one of the things that makes the Macintosh so easy to use is its consistency of interface. Your design goal is to help users find choices where they expect to find them, instead of having to hunt for them. You can find many helpful hints in *Human Interface Guidelines: The Apple Desktop Interface*.

Dialog layout guidelines

Generally, it should be easy for the user to see what information is required before the command can be run and what controls are currently selected.

When a script calls for nested dialog boxes, all required arguments, as well as the most frequent or useful arguments, should be on the first dialog box. In general, try to reduce the number of dialogs to a minimum. Ideally, the user should be able to see everything on one dialog, so that it is immediately clear from the dialog box which controls have been chosen.

The layout of controls within a dialog should correspond to the direction people normally read. Required arguments, if any, should be distinguished from optional arguments and presented in the first part of the first dialog page. The most important or frequently used arguments should follow after the required arguments. For example, in France people usually read left to right and top to bottom, so the layout of the dialog and controls should follow this pattern.

Use boxes to group similar items. Boxes can separate columns, portions of columns, or clusters of buttons. Boxes do not have to be labeled, though labels are often useful.

Buttons to select files or directories (or both) should be placed on the first dialog page when possible. Use the keywords `last1`, `last2`, and so on to permit this arrangement.

Normally, each dialog item corresponds to a single control or argument. In some cases, however, a command may have one or more commonly used group of controls. In these cases, some of the dialog items may correspond to control clusters. Note that the user should still be able to select all controls individually.

Use the keyword `string` if the possible values are infinite. If the number of values is a small, finite number, try to use radio buttons.

There are several standards for subdialog names:

- Subdialogs containing only Output and Error pop-up menus should be labeled "Output & Error."
- If a dialog contains only one subdialog of unrelated options, that subdialog should be labeled "More options." If the options are closely related, that relationship can be used to name the subdialog.
- If a dialog contains several subdialogs containing unrelated options, these subdialogs should be named "Options 1," "Options 2," and so on.

Dialog aesthetics

Try to avoid mixing control types (checkboxes, radio buttons, text boxes, and buttons). Try to make the dialog page look balanced. With few exceptions, dialogs look best with two columns per row. Use empty columns for spacing to prevent a column from appearing too wide.

Don't juxtapose unrelated sets of radio buttons. Remember that the first radio button in a cluster will be turned on by default. Take care to choose a default that is reasonable. It is often a good idea to add a button to a cluster of controls to represent the default action.

Descriptive information

The text associated with a dialog item should be understandable by the UNIX-naive user whenever possible. Options should be described in terms of the results that the user will see, rather than in terms of the underlying UNIX concepts.

Filename arguments should be specified by their function or role. For example, use "Files to be searched" rather than "Input."

Always try to show the effects of defaults. One example is to label the pop-up menus for output files "Output to" so that the default behavior is displayed on the screen.

Put useful information on the screen if it doesn't lead to clutter. For example, the UNIX command `date` takes as an argument a string formatted `mmddhhmm[yy]`. This format is small, useful, and easy to forget. It can be placed just above the text box where the user can refer to the format when entering the date. Examples of more extended information should be placed in the help message.

The help messages should expand on the text in the upper portion of the dialog box to provide information and, where possible, examples. Don't simply repeat the control text for the help message. If you can't think of anything else to add, rephrase the control text in case the user didn't understand the original text. When the user has to type in something, give examples of common usages.

Chapter 4 **dbx Reference**

This chapter describes the debugger `dbx`, a tool for source-level debugging and execution of C programs under A/UX.

The debugger operates by running the program being debugged as a child process. The debugger maintains control of the program being debugged by means of system hooks available through the `ptrace`(2) system call.

Using dbx

The dbx debugger can be used to symbolically debug all A/UX applications. It is particularly useful if the application makes calls to the A/UX Toolbox. The debugger can examine several kinds of code.

For specially compiled C code dbx can provide you with

- examination of the symbol table
- variable, expression, and condition tracing
- function and procedure tracing
- source-line tracing
- signal trapping
- variable assignment
- step-by-step execution
- variable- and expression -printing capabilities
- real-time editing capability

The debugger dbx also has the capability of examining object code at the machine-language level. These machine-level facilities of dbx can be used on any program. The ability to examine machine language is useful when you don't have the source code for a program, or when you want to inspect compiled assembly code to see exactly what the compiler and optimizer did to your source.

The other debuggers available with A/UX are

adb	This machine-level debugger is useful for debugging A/UX applications that do not call the A/UX Toolbox.
MacsBug	This symbolic debugger works on Macintosh applications. It is part of the Macintosh Programmer's Workshop, included with A/UX Developer's Tools.
sdb	This symbolic debugger is useful for debugging A/UX applications that do not call the A/UX Toolbox. Like dbx, sdb allows you to interact with the program at a source language level. Symbolic access to all variables is available, and procedures may be called directly from sdb. This debugger works on source code compiled with the A/UX compilers c89, cc, and f77.

dbx syntax

The command-line syntax for dbx is

dbx [-r] [-i] [-k] [-I *dir*] [-c *file*] [*objfile* [*coredump*]]

The *objfile* is an object file produced by the c89, cc, or f77 compilers. In order to use symbolic debugging, the object file must have been created using the -g compiler command option. Object files created with the -g option contain a symbol table that includes the names of the all the source files translated by the compiler to create it. These files are available for perusal while using the debugger. Files created without the -g option can be debugged, bu the symbol table inforamtion will not be available. Object files created with the -c option are intermediate relocatable object code files, which can be examined but not run. (Such files are called "dot-oh" files (.o) after the extension appended to the filename.)

- *Note:* Optimized code cannot be symbolically debugged with dbx; the code optimizer is disabled when the -g option of c89 is used.

If a file named core exists in the current directory or a *coredump* file is specified, dbx can be used to examine the state of the program when it faulted.

If the file .dbxinit exists in the current directory, the debugger commands in it are executed. dbx also checks for a .dbxinit in the user's home directory if there isn't one in the current directory.

The command line options and their meanings are

-c *file*	Execute the dbx commands in the file before reading from standard input.
-i	Force dbx to act as though standard input is a terminal.
-I *dir*	Add *dir* to the list of directories that are searched when looking for a source file. Normally dbx looks for source files in the current directory and in the directory where *objfile* is located. The directory search path can also be set within dbx with the use command.
-k	Map memory addresses.
-r	Execute *objfile* immediately. If it terminates successfully dbx exits. Otherwise, the reason for termination is reported and the user is offered the option of entering the debugger or letting the program fault. dbx will read from /dev/tty when -r is specified and standard input is not a terminal.

Unless `-r` is specified, `dbx` just prompts (using the prompt `>!!>`) and waits for a command.

Example

Consider the following example:

```
/*this is a C source code file, hello.c */
main()
      printf("Hello, world\n");
```

To use the debugger on this file, you first compile the file:

```
c89 -g hello.c -o hello.o
```

then invoke the debugger

```
dbx hello.o
```

Command list

Table 4-1 shows an alphabetic listing of the commands available with `dbx`. The following sections group the commands by function and describe them in detail.

- **Table 4-1** `dbx` commands

`alias`	`assign`	`call`	`catch`
`cont`	`delete`	`down`	`dump`
`edit`	`file`	`func`	`help`
`ignore`	`list`	`next`	`nexti`
`print`	`quit`	`rerun`	`return`
`run`	`set`	`sh`	`source`
`status`	`step`	`stepi`	`stop`
`stopi`	`trace`	`tracei`	`unalias`
`unset`	`up`	`use`	`whatis`
`where`	`whereis`	`which`	

Execution and tracing commands

You can use a variety of commands (discussed in the following list) to see how the program is flowing. Breakpoints can be set in several ways: `dbx` can stop at a certain source-line number, at a certain signal, when a procedure or function is called, when a variable is changed, or when a condition becomes true.

`run` [*args*] [<*filename*] [>*filename*]
`rerun` [*args*] [<*filename*] [>*filename*]
Start executing *objfile*, passing *args* as command-line arguments; the characters < and > can be used to redirect output and input in the usual manner. When `rerun` is used without any arguments, the previous argument list is passed to the program; otherwise it is identical to `run`. If *objfile* has been written since the last time the symbolic information was read in, `dbx` will read in the new information.

`trace` [`in` *procedure/function*] [`if` *condition*]
`trace` *source-line-number* [`if` *condition*]
`trace` *procedure/function* [`in` *procedure/function*] [`if` *condition*]
`trace` *expression* `at` *source-line-number* [`if` *condition*]
`trace` *variable* [`in` *procedure/function*] [`if` *condition*]
Have tracing information printed when the program is executed. A number is associated with the command that can be used with the `delete` command to turn the tracing off.

The first argument describes what is to be traced. If it is a source-line number, the line is printed immediately prior to being executed. Source-line numbers in a file other than the current one must be preceded by the name of the file in double quotation marks and a colon, for example,

```
"yoyodyne.c":21.
```

If the argument is a procedure or function name, every time it is called information is printed telling what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted. If the argument is a function, the value that *function* returns is also printed.

If the argument is an expression with an `at` clause, the value of the expression is printed whenever the identified source line is reached.

If the argument is a variable, the name and value of the variable is printed whenever it changes. Execution is substantially slower during this form of tracing.

If no argument is specified, all source lines are printed before they are executed. Execution is substantially slower during this form of tracing.

The clause "`in` *procedure/function*" restricts tracing information to be printed only while executing inside the given procedure or function.

The term *condition* is a boolean expression and is evaluated prior to printing the tracing information; if it is false then the information is not printed.

`stop if` *condition*
`stop at` *source-line-number* `[if` *condition*`]`
`stop in` *procedure/function* `[if` *condition*`]`
`stop` *variable*`[if` *condition*`]`
Stop execution when the given line is reached, procedure or function is called, variable is changed, or condition becomes true. Execution can be resumed with the `cont` command.

`status [>` filename`]`
Print the currently active `trace` and `stop` commands.

`delete` *command-number* ...
The traces or stops corresponding to the given numbers are removed. The numbers associated with traces and stops are printed by the `status` command.

`catch` *number*.
`catch` *signal-name*
`ignore` *number*.
`ignore` *signal-name*
Start or stop trapping a signal before it is sent to the program. This is useful when a program being debugged handles signals such as interrupts. A signal may be specified by number or by a name (for example, `SIGINT`). Signal names are case insensitive, and the `SIG` prefix is optional. By default all signals are trapped except `SIGCONT`, `SIGCHILD`, `SIGALRM`, and `SIGKILL`.

`cont` *integer*
`cont` *signal-name*
Continue execution from where it stopped. If a signal is specified, the process continues as though it had received the signal. Otherwise, the process is continued as though it had not been stopped.

Execution cannot be continued if the process has "finished," that is, if it has called `exit`. Even if this call has been made, however, the user can examine the program state because `dbx` does not allow the process to actually exit.

`step`
Execute one source line.

`next`

Execute up to the next source line. The difference between `next` and `step` is that if the line contains a call to a procedure or function, the `step` command will stop at the beginning of that block, while the `next` command will not.

`return` [*procedure*]

Continue until the named procedure is returned to, or until the current procedure returns if none is specified.

`call` *procedure*(*parameters*)

Execute the object code associated with the named procedure or function.

Example

Continuing the example defined previously, the debugger is now waiting for input. The following code sets a breakpoint at source line 3 and traces the value of the variable *whatnow* as it changes.

```
stop at 3
trace whatnow
run
```

The output looks like this:

```
whatever the output looks like until it stops
```

The program has stopped on source line 3, as requested. To finish running the program, you tell the program to continue:

```
cont
```

and the remainder of the output looks like this:

```
whatever the output looks like until it stops
```

Printing variables and expressions

Names are resolved first using the static scope of the current function, then using the dynamic scope if the name is not defined in the static scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message `[using` ***qualified name***`]` is printed. The name resolution procedure may be overridden by qualifying an identifier with a block name, for example, ***module.variable***. For C, source files are treated as modules named by the filename without the usual `.c` suffix.

Expressions are specified with an approximately common subset of C and Pascal syntax. Indirection can be denoted using either a prefix `*` or a postfix `^`; array subscripts are enclosed by brackets (`[ ]`). The field reference operator (`.`) can be used with pointers as well as records, making the C operator `->` unnecessary (although it is supported).

Types of expressions are checked; the type of an expression may be overridden by using *type-name* `(`*expression*`)`. When there is no corresponding named type, the special constructs `&`*type-name* and `$$`*tag-name* can be used to represent a pointer to a named type or C structure tag.

`assign` *variable* `=` *expression*
: Assign the value of the expression to the variable.

`dump` `[`*procedure*`]` `[>`*filename*`]`
: Print the names and values of variables in the given procedure, or the current one if none is specified. If the procedure given is ".", all the active variables are dumped.

`print` *expression* `[,` *expression* `...]`
: Print the values of the expressions.

`whatis` *name*
: Print the declaration of the given name, which may be qualified with block names as explained earlier in this section.

`which` *identifier*
: Print the full qualification of the given identifier, that is, the outer blocks with which the identifier is associated.

`up` `[`*count*`]`
`down` `[`*count*`]`
: Move the current function, which is used for resolving names, up or down the stack count levels. The default count is `1`.

`where`
: Print a list of the active procedures and functions. and the argument passed to them

`whereis` *identifier*
: Print the full qualification of all the symbols whose name matches the given identifier. The order in which the symbols are printed is not meaningful.

Example

An example using the above commands is

```
output listing
```

Accessing source files

/ regular expression [/]
? regular expression [?]
Search forward or backward in the current source file for the given pattern.

`edit` [*filename*]
`edit` *procedure/function-name*
Invoke an editor on *filename* or the current source file if no *filename* is specified. If a procedure or function name is specified, the editor is invoked on the file that contains it. Which editor is invoked by default depends on the installation. You can override the default by setting the environment variable `EDITOR` to the name of the desired editor.

`file` [*filename*]
Change the current source filename to *filename*. If none is specified, the current source file name is printed.

`func` [*procedure/function*]
Change the current function. If none is specified, print the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

`list` [*source-line-number* [, *source-line-number*]]
`list` *procedure/function*
List the lines in the current source file from the first line number to the second, inclusive. If no lines have been specified, the next `$listwindow` lines are listed (the default is 10). If the name of a procedure or function is given, lines *n-k* to *n+k* are listed, where *n* is the first statement in the procedure or function and *k* is defined by the value `$listwindow`.

`use` *directory-list*
Set the list of directories to be searched when looking for source files.

Example

An example using the above commands is

```
output listing
```

Command aliases and variables

`alias` *name name*
`alias` *name string*
`alias` *name* (*parameters*) *string*

When commands are processed, `dbx` first checks to see if the word is an alias for either a command or a string. If it is an alias, then `dbx` treats the input as though the corresponding string (with values substituted for any parameters) had been entered. For example, to define an alias `rr` for the command `rerun`, you can say

```
alias rr rerun
```

To define an alias called `b` that sets a stop at a particular line, you can say

```
alias b(x) "stop at x"
```

The command `b(12)` will subsequently expand to `stop at 12`.

`set` *name* [`=` *expression*]

The `set` command defines values for debugger variables. The names of these variables cannot conflict with names in the program being debugged, and are expanded to the corresponding expression within other commands. The following variables have a special meaning:

`$frame`	Setting this variable to an address causes `dbx` to use the stack frame pointed to by the address for doing stack traces and accessing local variables.

`$hexchars`
`$hexints`
`$hexoffsets`
`$hexstrings`
	When these variables are set, `dbx` prints out characters, integers, offsets from registers, or character pointers, respectively, in hexadecimal.

`$listwindow`	The value of this variable specifies the number of lines to list around a function or when the list command is given without any parameters. Its default value is 10.
`$mapaddrs`	Setting (unsetting) this variable causes `dbx` to start (stop) mapping addresses.

`$unsafecall`
`$unsafeassign`
: When `$unsafecall` is set, strict type checking is turned off for arguments to subroutine calls or function calls (for example, in the `call` statement). When `$unsafeassign` is set, strict type checking between the two sides of an assign statement is turned off. These variables should be used only with great care, because they severely limit the usefulness of `dbx` for detecting errors.

`unalias` ***name***
: Remove the alias for ***name***.

`unset` ***name***
: Delete the debugger variable associated with ***name***.

Example

An example using the above commands is

```
output listing
```

Machine-level commands

`tracei` [*address*] `[if` *cond*]
`tracei` [*variable*] `[at` *address*] `[if` *cond*]
`stopi` [*address*] `[if` *cond*]
`stopi [at]` [*address*] `[if` *cond*]
Turn on tracing or set a stop using a machine instruction address.

`stepi`
`nexti` Execute a single step as in `step` or `next`, but do a single instruction rather than a source line. .i.`stepi` command 4-; .i.`nexti` command 4-;

address , *address*/ [*mode*]
address / [*count*] [*mode*]
Print the contents of memory starting at the first address and continuing up to the second address or until *count* items are printed. If the address is ".", the address following the one printed most recently is used. The mode specifies how memory is to be printed; if it is omitted the previous mode specified is used. The initial mode is `x`. The following modes are supported:

- `i` Print the machine instruction.
- `d` Print a short word in decimal.
- `D` Print a long word in decimal.
- `o` Print a short word in octal.
- `O` Print a long word in octal.
- `x` Print a short word in hexadecimal.
- `X` Print a long word in hexadecimal.
- `b` Print a byte in octal.
- `c` Print a byte as a character.
- `s` Print a string of characters terminated by a null byte.
- `f` Print a single-precision real number.
- `g` Print a double-precision real number.

Symbolic addresses are specified by preceding the name with an `&`. Registers are denoted by `$r`*n* where *n* is the number of the register. Addresses may be expressions made up of other addresses and the operators +, -, and indirection (unary *).

`help` Print out a synopsis of `dbx` commands.

`quit` Exit `dbx`.

`sh` *command-line*
Pass the command line to the shell for execution. The `SHELL` environment variable determines which shell is used.

`source` *filename*
Read `dbx` commands from the given filename.

Example

An example using the above commands is

```
output listing
```

Warnings

`dbx` suffers from the same "multiple include" malady as did `sdb`. If you have a program consisting of a number of object files and each is built from source files that include header files, the symbolic information for the header files is replicated in each object file. Since about one debugger start-up is done for each link, having the linker (`ld`) reorganize the symbol information would not save much time, though it would reduce some of the disk space used.

This problem is an artifact of the unrestricted semantics of `#include` files in C; for example, an include file can contain static declarations that are separate entities for each file in which they are included.

Some problems remain with support for individual languages. Fortran problems include inability to assign to logical, complex and double complex variables; inability to represent parameter constants that are not type integer or real; peculiar representation for the values of dummy procedures (the value shown for a dummy procedure is actually the first few bytes of the procedure text; to find the location of the procedure, use `&` to take the address of the variable).

Chapter 5 `c89` Command Syntax

This chapter describes the command syntax for `c89`. The `c89` command is a program that invokes the C compiler, assembler, and loader, as appropriate. (The default is to invoke each one in turn.) The compiler incorporates a preprocessor phase that can be directed to output a preprocessed version of the source file. This capability is useful for checking the incorporation of `#include` files and the substitution of manifest constants.

Command syntax

The syntax for `c89` is

`c89 [`*commandopt*`...] `*file*`...`

where *commandopt* is 0 or more command options (see the section "Options") and *file* is one or more filenames. Command options and filenames can be mixed; the only position-dependent option is `-l`, which specifies libraries to be searched by the loader.

`c89` recognizes filenames of the form

file.x

The two-character extension *.x* identifies the contents of the file, as shown in Table 5-1. A filename with no extension is assumed to be a library archive.

■ **Table 5-1** Extension conventions

Extension	Contents	Example
`.c`	C source code	`program.c`
`.i`	Preprocessor output	`program.i`
`.s`	Assembler source	`program.s`
`.o`	Assembler output	`program.o`
`.a`	Library archive	`libc.a`

Default behavior

Running `c89` with no command options on a file named *file*`.c` invokes the C compiler, the assembler, and the loader in turn. This process produces an executable file in the current directory; by default this executable file is named `a.out`. The C compiler includes normal UNIX preprocessing functions.

`c89` has a large number of command options that can be used to control the compilation process. In addition, other command options can be passed to the preprocessor phase of the compiler, and the compiler, assembler, and loader. The sections that follow describe these command options.

- *Note:* In order to run the optimizer, a command option must be set. The optimizer produces significant performance improvement. However, optimized code cannot be symbolically debugged.

Feature test macros

A/UX defines the following feature test macros:

`_AUX_SOURCE`

`_BSD_SOURCE`

`_SYSV_SOURCE`

`_POSIX_SOURCE`

The feature test macros `_SYSV_SOURCE` and `_BSD_SOURCE` represent the historical implementations on which A/UX is based. `_AUX_SOURCE` represents extensions to the historical implementations that are specific to A/UX. `_POSIX_SOURCE` is not normally defined.

POSIX specifies certain symbols that are defined in header files. Some of these header files may also define symbols in addition to those defined by POSIX, potentially conflicting with symbols defined by an application program. Feature test macros control the visibility of these symbols in the header files required by POSIX. When POSIX compilation is selected (with the `-Zp` option) test macros other than `_POSIX_SOURCE` are not defined.

Another test macro `_AUX_C_EXTENSIONS` is visible provided the compilation is not done in strict ANSI mode (the `-Xc` option). This allows programs to make use of the A/UX extensions such as direct functions and the `pascal` type qualifier.

Options

All options recognized by the `c89` command are listed and described in the sections that follow. Some options are used by the compiler itself, some are passed on to the assembler (`as`), and some are passed on to the loader (`ld`).

Options recognized and executed by c89

Table 5-2 lists the options used to control the behavior of the compiler.

■ **Table 5-2** Options executed by c89

Option	Argument	Description
-A	*factor*	Expands the default symbol table allocations for the assembler and loader. The default allocation is multiplied by the factor given.
-B	*string*	Construct pathnames for the substitute preprocessor, compiler, and loader passes by concatenating *string* with the suffixes `acomp`, `newoptim`, `as`, and `ld`. The passes affected may be specified by the `-t` option.
-E	none	Same as the `-P` option except output is directed to the standard output.
-H	none	Print, one per line, the pathname of each file included during the compilation on the standard output.
-O	none	This option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. When the optimizer is used, line numbers used for symbolic debugging may be transposed.
-P	none	Suppress compilation and loading; that is, invoke only the preprocessor phase of the compiler and leave the output on corresponding files with the extension `.i`.
-R	none	Have assembler remove its input file when done.
-S	none	Compile the named C programs and leave the assembly language output within corresponding files suffixed `.s`.
-V	none	Cause each invoked tool to print its version information on the standard output. Also causes the compiler to issue additional warnings for possible `lint`-type errors.
-X	*flag*	Specify the degree of conformance to the ANSI C standard. The flag can be one of the following: `a` (ANSI). The compiled language includes all the new features of ANSI C. The compiler warns about language constructs that have differing behavior between the new and old versions and uses the new interpretation of constructs with differing behavior. This includes, for example, warning about the use of trigraphs, the new escape sequence, and changes to the integral promotion rules.

(continued)

■ **Table 5-2** Options executed by `c89` (continued)

Option	Argument	Description
		`c` (conformance). The compiled language and associated header files conform to the ANSI C standard, but include all conforming extensions of `-Xa`. Warnings will be produced about some of these extensions. Only ANSI-defined identifiers are visible in the standard header.
		`t` (transition). The compiled language includes all new features compatible with older (pre-ANSI) C languages. The compiler warns about all language constructs that have differing behavior between the new and old versions, and uses pre-ANSI C interpretation. This is the default behavior.
`-W`	`c`, *arg1*[,*arg2*...]	Pass the argument(s) *arg1 to c*, where *c* is one of `[02al]`, indicating compiler, optimizer (`2`), assembler (`a`), or loader (`l`), respectively.
`-Z`	*flags*	Special *flags* to override the default behavior (see `c89`(1)). Currently recognized *flags* are:
		`l` Suppress selection of a loader command file.
		`t` Do not delete temporary files.
		`P` Compile for the A/UX POSIX environment. Load the file with a library module that calls `setcompat`(2) with the `COMPAT_POSIX` flag set. Define only the `_POSIX_SOURCE` feature test macro.
		`S` Compile to be SVID compatible. Load the file with a library module that calls `setcompat`(2) with the `COMPAT_SVID` flag set. Define only the `_SVSV_SOURCE` feature test macro.
		`B` Compile to be BSD compatible. Load the file with a library module that calls `setcompat`(2) with the `COMPAT_BSD` flag set. Define only the `_BSD_SOURCE` feature test macro.
`-#`	none	Special debug option that, without actually starting the program, echoes the names and arguments of subprocesses that would have started.
`-a`	none	Include source code as comments in the assembly file generated with the `-S` option.
`-c`	none	Suppress the loading phase of compilation and force production of a relocatable object file even if only one file is compiled.
`-f`	`m68881`	This option is ignored.

(continued)

■ **Table 5-2** Options executed by `c89` (continued)

Option	Argument	Description
`-g`	none	This option produces information for a symbolic debugger. (For more information, see Chapter 4, "`dbx` Reference.")
`-n`	none	Arrange for the loader to produce an executable file that is loaded in such a manner that the text can be made read-only and shared (nonvirtual) or paged (virtual).
`-p`	none	Arrange for the compiler to produce code that counts the number of times each routine is called. A `mon.out` file will be produced at normal termination of execution of the object program. See `prof`(1) for usage of the `mon.out` file.
`-t`	`[02al]`	Find only the designated compiler (`0`), optimizer (`2`), assembler (`a`), and loader (`l`) passes whose names are constructed with the *string* argument to the `-B` option. In the absence of a `-B` option and its argument, *string* is taken to be `/lib/n`.
`-v`	none	Print the command line for each subprocess executed.

Options recognized by `c89` and passed to `as`

Table 5-3 lists the options that are recognized by the compiler and passed to the assembler.

■ **Table 5-3** Options passed to `as`

Option	Argument	Description
`-68030`	none	Directs the assembler to recognize the memory management unit (MMU) instructions for a Motorola 68030 microprocessor.
`-68851`	none	Directs the assembler to recognize the coprocessor instructions for a Motorola 68851 PMMU.

Options recognized by `c89` and passed to `ld`

Table 5-4 lists the options that are recognized by the compiler and passed to the loader.

■ **Table 5-4** Options passed to `ld`

Option	Argument	Description
`-l`	*name*	Same as `-l` in `ld`(1). Search a library `lib`*name*`.a`, where *name* is up to seven characters. A library is searched when its name is encountered, so the placement of `-l` is significant. By default, libraries are located in `LIBDIR`. If you plan to use the `-L` option, that option must precede `-l` on the command line.
`-o`	*outfile*	Same as `-o` in `ld`(1). Produce an output object file, *outfile*. The default name of the object file is `a.out`.
`-s`	none	Same as `-s` in `ld`(1). Strip the line number entries and symbol table information from the output of object file.
`-L`	*dir*	Same as `-L` in `ld`(1). Search for `lib`*name*`.a` in the named *dir* before looking in `LIBDIR`. This option is effective only if it precedes the `-l` option on the command line.
`-V`	none	Print the version of the loader that is invoked.

For more information on any of the options which `c89`(1) passes to the loader `ld`(1), see the *A/UX Command Reference* and chapter 6, "The `ld` loader."

Options recognized by `c89` and passed to the preprocessor

Table 5-5 lists the options that are recognized by the compiler and used to modify its preprocessor phase.

■ **Table 5-5** Options passed to the preprocessor

Option	Argument	Description
`-C`	none	All comments are passed along. The default strips out all comments.
`-D`	*symbol*[=*def*]	Define the external *symbol* and give it the value *def* (if specified). If no *def* is given, *symbol* is defined as 1.

(continued)

■ **Table 5-5** Options passed to the preprocessor (continued)

Option	Argument	Description
`-I`	*dir*	Search for `#include` files that do not begin with `/` in the named *dir* before looking in the directories on the standard list. Thus, `#include` files whose names are enclosed in quotation marks (for example, `#include "thisfile"`) are first searched for in the directory of the file being compiled, then in directories named by the `-I` options, and last in directories on the standard list. For `#include` files whose names are enclosed in <> (for example, `#include <thisfile>`), the directory of the file being compiled is not searched.
`-U`	*symbol*	Remove any initial definition of *symbol* ("undefine" *symbol*), where *symbol* is a reserved name that is predefined by the particular preprocessor.

Intermediate output

By using appropriate options, collected in Table 5-6, you can terminate compilation early to produce one of several intermediate translations. A number of such options are available:

■ **Table 5-6** Intermediate output options

Option	Description
`-c`	This option produces relocatable object files. It is often desirable to use the `-c` option to save relocatable files so that changes to one file do not then require recompilation of other files. A separate call to `c89`, with the relocatable files but without the `-c` option, creates the loaded executable `a.out` file. A relocatable object file created under the `-c` option has the same root as the relocatable object file, but the extension is `.o` instead of `.c`.
`-E`	This option gives roughly the same output as `-P`, except the output goes to `stdout`.
`-S`	This option produces assembly-source expansions for C code.

`-P` This option produces the output of the preprocessor. When you use this option, the compilation process stops after preprocessing. Output from the preprocessor is left in an output file with the extension `.i` (for example, `file1.i`). A preprocessed output file can be subsequently processed by `c89`, but only if its filename is changed to one with the extension `.c`. Except for those produced by the preprocessor, any intermediate files may be saved and resubmitted to the `c89` command, with other files or libraries included as necessary.

`-W` This option lets you specify options for each step that is normally invoked from the `c89` command line, that is, (1) preprocessing, (2) the first pass of the compiler, (3) the second pass of the compiler, (4) optimization, (5) assembly, and (6) loading.

At this time, only assembler and loader options can be used with the `-W` option. The most common example of the `-W` option is

`-Wl,-VS,`*n*

which passes the `-VS`*n* option to the loader (`ld`(1)). In the example

`-Wa,`*-option*

the compiler will pass the *-option* to the assembler.

`-O` This option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. When the optimizer is used, line numbers used for symbolic debugging may be transposed.

`-g` This option produces information for a symbolic debugger.

Chapter 6 **as Reference**

Programmers familiar with the MC68000 family of processors should be able to debug code produced by the A/UX resident assembler, `as`, after reviewing this chapter, but this is not a reference for the processor itself. Details about the effects of instructions, meaning of status register bits, handling of interrupts, and many other issues are not dealt with here. This chapter should therefore be used in conjunction with the following reference manuals:

- *MC68020 32-Bit Microprocessor User's Manual* (Prentice-Hall, 1984.)
- *MC68030 Enhanced 32-Bit Microprocessor User's Manual Second Edition* (Prentice-Hall, 1989.)
- *MC68851 Paged Memory Management Unit User's Manual*, (Motorola, Inc., 1985.)
- *MC68881 Floating Point Coprocessor User's Manual*, (Motorola, Inc., 1985.)

Warnings

A few important warnings to the `as` user should be emphasized at the outset. Although, for the most part, there is a direct correspondence between `as` notation and the notation used in the documents listed in the introduction to this chapter, several exceptions could lead the unsuspecting user to write incorrect code. In addition to the exceptions described in the following paragraphs, refer also to the sections "Address Mode Syntax" and "Machine Instructions" later in this chapter for further information.

Comparison instructions

The order of the operands in compare instructions follows one convention in the *MC68020* and *MC68030 Programmer's Reference Manuals* and the opposite convention in `as`. Using the convention of the *MC68020 Programmer's Reference Manual*, you might write

```
CMP.W D5, D3        # Is D3 less than D5?
BLE   IS_LESS       # Branch if less.
```

Using the `as` convention, you would write

```
cmp.w %d3,%d5       # Is d3 less than d5 ?
ble   is_less       # Branch if less.
```

The convention used by `as` makes for straightforward reading of compare and branch instruction sequences, with this exception: if a compare instruction is replaced by a subtract instruction, the effect on the condition codes is entirely different. This result may be confusing to programmers who are used to thinking of a comparison as a subtraction whose result is not stored. Users of `as` who become accustomed to its convention find that both the compare and subtract notations make sense in their respective contexts.

Case sensitivity

In the A/UX implementation, only lowercase instruction and register names are valid. For example,

```
mov   %d1,%d2     # works
```

is acceptable, while

```
MOV   %D1,%D2     # does not work
```

is not. This is especially important for those who wish to port existing code from other machines.

Overloading of opcodes

Another issue that users must be aware of arises from the MC68000-family processors' use of several different instructions to do more or less the same thing. For example, the *MC68020 Programmer's Reference Manual* lists the instructions SUB, SUBA, SUBI, and SUBQ, which all have the effect of subtracting their source operand from their destination operand. `as` replaces the separate `suba`, `subi`, and `subq` instructions, allowing all these operations to be specified by a single assembly instruction `sub`. On the basis of the operands given to the `sub` instruction, the `as` assembler selects the appropriate MC68000-family operation code. The danger created by this convenience is that it could give the misleading impression that all forms of the SUB operation are semantically identical. In fact, they are not. The careful reader of the *MC68020 Programmer's Reference Manual* will notice that whereas SUB, SUBI, and SUBQ all affect the condition codes in a consistent way, SUBA does not affect the condition codes at all. Consequently, the `as` user must be aware that when the destination of a `sub` instruction is an address register (which causes the `sub` to be mapped into the operation code for `suba`), the condition codes will not be affected.

Using `as`

The A/UX command `as` invokes the assembler and has the following syntax:

`as [-m] [-n] [-o` *outfile*`] [-R] [-V] [-A` *factor*`]` *filename*

The command options listed in Table 6-1 may be specified in any order.

■ **Table 6-1** Options to `as`

Option	Description
`-A` *factor*	Expand the default symbol table by the factor given.
`-R`	Remove (unlink) the input file after assembly is completed. This command option is off by default.

(Continued)

■ **Table 6-1** Options to `as` (Continued)

Option	Description
`-v`	Write the version number of the assembler being run on the standard error output.
`-m`	Run the `m4` macro preprocessor on the input to the assembler. ◆ *Note:* If the `-m` command option is used, keywords for `m4` cannot be used as identifiers (variables, functions, labels, and so on) in the input file because `m4` cannot determine which are assembler symbols and which are real `m4` macros.
`-n`	Turn off long/short address optimization. By default, address optimization takes place.
`-o` *outfile*	Put the output of assembly in *outfile*. By default, the output filename is formed by removing the `.s` suffix, if there is one, from the input filename and appending an `.o` suffix.
`-68030`	Assemble for the MC68030 processor. This gives you access to an enhanced feature set as compared to the default MC68020 assembly, but the code will not run on all Macintosh II models.
`-68851`	Assemble for the MC68851 Memory Management Unit (MMU). This command option is on by default.

General syntax rules

The following sections discuss the components of the assembly language produced by the `as` assembler.

Format of assembly language code

Typical lines of `as` assembly code look like these:

```
# Clear a block of memory at location %a3
   text     2
   mov.w    &const,%d1
loop:       clr.l    (%a3)+
   dbf      %d1,loop          # go back for const
                              # repetitions
init2:
   clr.l count;  clr.l credit;  clr.l debit;
```

where the suffix to `clr` is always the letter `l` (ell), while `%d1` indicates data register 1 (one).

These general points about the example should be noted:

- An identifier occurring at the beginning of a line and followed by a colon (`:`) is a **label.** In the example above, `loop` and `init2` are labels. One or more labels may precede any assembly language instruction or pseudo-operation. Refer to the section "Location Counters and Labels."
- A line of assembly code need not include an instruction. It may consist of a comment alone (introduced by `#`), or a label alone (terminated by `:`), or it may be entirely blank.
- It is good practice to use tabs to align assembly-language operations and their operands into columns, but this is not a requirement of the assembler. An opcode may appear at the beginning of the line, if desired, and spaces may precede a label. A single blank or tab suffices to separate an opcode from its operands. Additional blanks and tabs are ignored by the assembler.
- It is permissible to write several instructions on one line, separating them by semicolons. The semicolon is syntactically equivalent to a newline character; however, a semicolon inside a comment is ignored.

Comments

Comments are introduced by the character `#` and continue to the end of the line. Comments may appear anywhere and are disregarded by the assembler.

Identifiers

An identifier is a name for a variable, label, register, or function. It consists of a string of characters taken from the set `a-z`, `A-Z`, `_`, `~`, `%`, and `0-9`. The first character of an identifier must be a letter (uppercase or lowercase) or an underscore. Uppercase and lowercase letters are distinguished; for example, `con35` and `CON35` are two distinct identifiers.

Identifiers can be up 1024 characters long (this limit is imposed by the loader).

The value of an identifier is established by the `set` pseudo-operation (refer to the section "Symbol Definition Operations") or by using it as a label (refer to the section "Location Counters and Labels").

The tilde character (~) has special significance to the assembler. A tilde used alone as an identifier means "the current location." A tilde used as the first character in an identifier becomes a period (.) in the symbol table. This is provided for backward compatibility. The assembler now directly supports symbols such as `.eos` and `.0fake` to be entered into the symbol table, as required by the Common Object File Format. Information about file formats is provided in Section 4 of the *A/UX Programmer's Reference*.

Register identifiers

A register identifier is an identifier preceded by the character `%`, and represents one of the MC68000-family processors' registers. The predefined MC68020 register identifiers are shown in Table 6-2.

■ **Table 6-2** Predefined MC68020 registers

Name	Description
`%d0-7`	Data registers
`%a0-5`	Address registers
`%a6`	Address register (also defined as `%fp`)
`%a7, %usp`	User stack pointer (also defined as `%sp`)
`%pc`	Program counter
`%ccr`	Condition code register
`%isp`	Interrupt stack pointer
`%msp`	Master stack pointer
`%sr`	Status register
`%vbr`	Vector base register
`%sfc`	Alternate function code register (also defined as `%sfcr`)
`%dfc`	Alternate function code register (also defined as `%dfcr`)

`%cacr`	Cache control register
`%caar`	Cache address register

To preserve the upward compatibility of MC68000 code, the identifiers `%a7`, `%sp`, and `%usp` represent the same machine register. Likewise, `%a6` and `%fp` are equivalent. Use of both `%a7` and `%sp`, or `%a6` and `%fp`, in the same program may result in confusion and should be avoided. The registers `%sfc` and `%sfcr` are also equivalent, as are `%dfc` and `%dfcr`.

The entire register set of the MC68000 and MC68010 is included in the MC68020 register set. Table 6-3 shows the new control registers for the MC68030:

■ **Table 6-3** Additional registers for MC68030

Name	Description
`%crp`	CPU root pointer
`%srp`	Supervisor root pointer
`%tc`	Translation control register
`%tt0, %tt1`	Translation registers
`%psr`	MMU status register

Various registers can be suppressed; these suppressed registers (also called *zero registers*) are used in various complex MC68020 addressing modes. The notation for suppressed registers is `%zd`*n* for data register *n*, `%za`*n* for data register *n*, and `%zpc` for the suppressed program counter.

Constants

`as` deals only with integer constants. They may be entered in decimal, octal, or hexadecimal, or they may be entered as character constants. Internally, `as` treats all constants as 32-bit binary 2's-complement quantities.

Numeric constants

A decimal constant is a string of digits beginning with a nonzero digit. An octal constant is a string of digits beginning with zero. A hexadecimal constant consists of the characters `0x` or `0X` followed by a string of characters from the set `0-9, a-f,` and `A-F`. In hexadecimal constants, uppercase and lowercase letters are not distinguished. Here are some examples:

```
set   const,35     # decimal 35
mov.w &035,%d1     # octal 35 (decimal 29)
set   const, 0x35  # hex 35 (decimal 53)
mov.w &0xff,%d1    # hex ff (decimal 255)
```

Character constants

An ordinary character constant consists of a single quotation mark (') followed by an arbitrary ASCII character other than the backslash (\). The value of the constant is equal to the ASCII code for the character. For example, the character constant `'A` has value `0x41`. Special meanings of characters are overridden when used in character constants; for example, if `'#` is used, the # is not treated as introducing a comment.

Special character constants convey special information to the assembler. A special character constant consists of `'\` followed by another character. All the special character constants are listed in Table 6-4.

■ **Table 6-4** Special character constants

Constant	Value	Meaning
`'\b`	`0x08`	Backspace
`'\t`	`0x09`	Horizontal tab
`'\n`	`0x0a`	Newline (line feed)
`'\v`	`0x0b`	Vertical tab
`'\f`	`0x0c`	Form feed
`'\r`	`0x0d`	Carriage return
`'\\`	`0x5c`	Backslash

Other syntactic details

A discussion of expression syntax appears in the section "Expressions." Information about the syntax of specific components of `as` instructions and pseudo-operations is given in the sections "Pseudo-operations," "Span-Dependent Optimization," and "Address Mode Syntax."

Segments, location counters, and labels

The following sections describe how the assembler arranges and locates various pieces of code.

Segments

A program in `as` assembly language can be broken into segments known as `text`, `data`, and `bss` segments. The convention regarding the use of these segments is to place instructions in `text` segments, initialized data in `data` segments, and uninitialized data in `bss` segments. The assembler does not enforce this convention, however. For example, it permits intermixing of instructions and data in a `text` segment if specifically directed to mix the segments. Routines to be placed in the shared library may also have an `init` segment, which contains initialization fragments. An `init` segment is treated similarly to a `text` segment.

This convention of using separate `text`, `data`, and `bss` segments permits the sharing of `text` segments between programs on systems utilizing shared memory, such as A/UX. If several copies of a program are running at once, as can happen when users are logged on over a network, there is only one instance of the text segment in memory, thus conserving memory space.

Primarily to simplify compiler code generation, the assembler permits up to four separate `text` segments and four separate `data` segments named `0`, `1`, `2`, and `3`. The assembly-language program may switch freely among them by using assembler pseudo-operations (see the section "Location Counter Control Operations" later in this chapter). This can be handy, for example, if you want to put all the constants in one segment and all the function in another. When generating the object file, the assembler concatenates the `text` segments to generate a single `text` segment, and the `data` segments to generate a single `data` segment. Thus, the object file contains only one `text` segment and only one `data` segment. There is always only one `bss` segment, and it maps directly into the object file.

Because the assembler keeps together everything from a given segment when generating the object file, the order in which information appears in the object file may not be the same as in the assembly-language file. For example, if the data for a program consists of

```
data  1      # segment 1
short 0x1111
data  0      # segment 0
long  0xffffffff
data  1      # segment 1
byte  0xff
```

the assembler groups the data for segment 0 together (in the order the assembler encounters it), then groups the data for segment 1. It then places the data for segment 1 after the data for segment 0 as it builds the object file. Thus, for the example just given, equivalent object code is generated by

```
data  1
data  0
long  0xffffffff
short 0x1111
byte  0xff
```

In this equivalent code example, the first statement

```
data  1
```

is effectively ignored.

Location counters and labels

The assembler maintains separate location counters for the `bss` segment and for each of the `text` and `data` segments. The **location counter** for a given segment is incremented by 1 for each byte generated in that segment.

The location counters allow values to be assigned to labels. When an identifier is used as a label in the assembly-language input, the value of the current location counter is assigned to the identifier. The assembler also keeps track of the segment in which the label appeared. Thus, the identifier represents a memory location relative to the beginning of a particular segment. Any label relative to the location counter should be within the text segment.

Types

Identifiers and expressions can have values of different types.

- In the simplest case, an expression or identifier may have an **absolute value**, such as 29, -5000, or 262143.

◆ *Note:* The term absolute value is not used here in the mathematical sense.

- An expression or identifier may have a value relative to the start of a particular segment. Such a value is known as a relocatable value. The memory location represented by such an expression cannot be known at assembly time, but the relative values of two such expressions (that is, the difference between them) can be known if they refer to the same segment.
- Identifiers that appear as labels have relocatable values.
- If an identifier is never assigned a value, it is assumed to be an undefined external. Such identifiers may be used with the expectation that their values will be defined in another program and therefore known at load time, but the relative values of undefined externals cannot be known.

Expressions

Since the value of some expressions cannot be known at assembly time, expression involving such values may also not be known at assembly time. This section provides the rules for determining if expressions involving unknown values are knowable.

For conciseness, the following abbreviations are useful:

abs	absolute expression
rel	relocatable expression
ext	undefined external

All constants are absolute expressions. An identifier may be thought of as an expression having the identifier's type. Expressions may be built up from lesser expressions using the operators +, -, *, and /, according to the following type rules:

abs + *abs* = *abs*

abs + *rel* = *rel* + *abs* = *rel*

abs + *ext* = *ext* + *abs* = *ext*

abs - *abs* = *abs*

rel - *abs* = *rel*

ext - *abs* = *ext*

rel - *rel* = *abs* (provided that the two relocatable expressions are relative to the same segment)

abs * *abs* = *abs*

abs / *abs* = *abs*

−abs = *abs*

rel − *rel* expressions are permitted only within the context of a switch statement (see the section "Switch Table Operation" later in this chapter). Use of a *rel* − *rel* expression is dangerous, particularly when dealing with identifiers from `text` segments. The problem is that the assembler will determine the value of the expression before it has resolved all questions concerning span-dependent optimizations.

The unary minus operator takes the highest precedence; the next highest precedence is given to * and /, and lowest precedence is given to + and binary −. Parentheses may be used to coerce the order of evaluation.

If the result of a division is a positive noninteger, it will be truncated toward zero. If the result is a negative noninteger, the direction of truncation cannot be guaranteed.

Pseudo-operations

This section details instructions to the assembler that do not involve expressions or operators. Collectively these are known as pseudo-operations. Any pseudo-operation can be prefixed by a period (.); this is provided for backward compatibility.

Data initialization operations

The following pseudo-operations allocate memory space for a program.

`byte` *abs, abs, ...*

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive bytes in the assembly output.

`short` *abs, abs, ...*

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive 16-bit words in the assembly output.

`long` *expr, expr, ...*

One or more expressions, separated by commas, may be given. Each expression may be absolute, relocatable, or undefined external. A 32-bit quantity is generated for each such argument (in the case of relocatable or undefined external expressions, the actual value may not be filled in until load time). Alternatively, the arguments may be bitfield expressions. A bitfield expression has the form

n`:` *value*

where both *n* and *value* denote absolute expressions. The quantity *n* represents a field width; the low-order *n* bits of *value* become the contents of the bitfield. Successive bitfields fill up 32-bit `long` quantities, starting with the high-order part. If the sum of the lengths of the bitfields is less than 32 bits, the assembler creates a 32-bit `long` with 0s filling out the low-order bits. For example,

```
long    4: -1, 16: 0x7f,  12:0, 5000
```

and

```
long    4: -1, 16: 0x7f, 5000
```

are equivalent to

```
long    0xf007f000, 5000
```

as shown in Figure 6-1.

Bitfields may not span pairs of 32-bit longs. Thus,

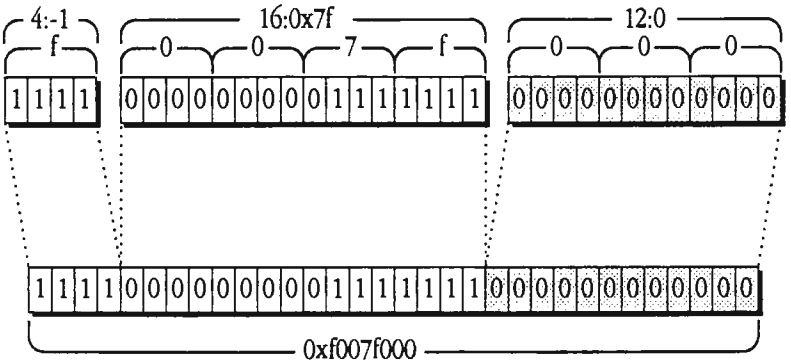
```
long    24: 0xa, 24: 0xb, 24:0xc
```

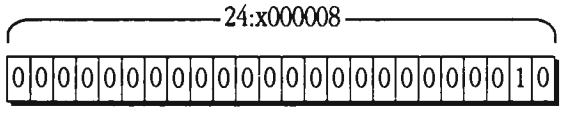
yields the same result as

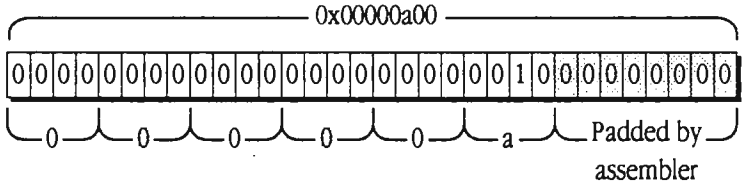
```
long    0x00000a00, 0x00000b00, 0x00000c00
```

as shown in Figure 6-1.

■ **Figure 6-1** Bitfield concatenation







`space` *abs* The value of *abs* is computed, and the resultant number of bytes of 0 data is generated. For example,

```
space    6
```

is equivalent to

```
byte    0,0,0,0,0,0
```

Symbol definition operations

The following pseudo-operations allocate memory space for a program:

`set` *identifier, expr*

The value of *identifier* is set equal to *expr*, which may be absolute or relocatable.

`comm` *identifier, abs*

The named *identifier* is assigned to a common area of size *abs* bytes. If *identifier* is not defined by another program, the loader will allocate space for it.

`lcomm` *identifier, abs*

The named *identifier* is assigned to a local common area of size *abs* bytes. This results in allocation of space in the `bss` segment. The type of *identifier* becomes relocatable.

`global` *identifier* This causes *identifier* to be externally visible. If *identifier* is defined in the current program, declaring it `global` allows the loader to resolve references to *identifier* in other programs. If *identifier* is not defined in the current program, the assembler expects an external resolution.

Location counter control operations

The following pseudo-operations define the segment in which code is to be allocated, and the alignment within that segment.

`data` *abs* The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the `data` segment into which assembly is to be directed. If no argument is present, assembly is directed into `data` segment 0.

`text` *abs* The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the `text` segment into which assembly is to be directed. If no argument is present, assembly is directed into `text` segment 0. Before the first `text` or `data` operation is encountered, assembly is by default directed into `text` segment 0.

`org` *expr* The current location counter is set to *expr*, which must represent a value in the current segment and must not be less than the current location counter.

`even` The current location counter is rounded up to the next even value.

`longeven` The current location counter is rounded up to the next 4-byte multiple value.

`align` *n*	The current location counter is rounded to a multiple of *n* bytes, where *n* can be 2,4,8,or 16. `even` is equivalent to `align 2`; `longeven` is equivalent to `align 4`.
`init`	The assembly is directed into the `init` segment. This operation is typically used for shared library initialization fragments.

Symbolic debugging operations

The assembler allows for symbolic debugging information to be placed into the object code file with special pseudo-operations. The information typically includes line numbers and information about C language symbols, such as their type and storage class. The C compiler `c89` generates symbolic debugging information when the `-g` command option is used. Assembler programmers may also include such information in source files.

`file` and `ln`

The `file` pseudo-operation passes the name of the source file into the object file symbol table. It has the form

`file` "*filename*"

where *filename* must be enclosed in straight double quotation marks.

The `ln` pseudo-operation makes a line number table entry in the object file; that is, it associates a line number with a memory location. Usually the memory location is the current location in text. The format is

`ln` *line* [,*value*]

where *line* is the line number. The optional value is the address in a `text`, `data`, or `bss` segment to associate with the line number. The default when *value* is omitted (which is usually the case) is the current location in `text`.

Symbol attribute operations

The basic symbolic testing pseudo-operations are `def` and `endef`. These operations enclose other pseudo-operations that assign attributes to a symbol and must be paired. The basic syntax for using `def` and `endef` is

`def` *name*
attrasgn
attrasgn

.
.
.
`endef`

where *attrasgn* may be any one of the attribute assigning operations shown in the list at the end of this section.

The term `def` does not define the symbol, although it does create a symbol table entry. Because an undefined symbol is treated as external, a symbol that appears in a `def` pseudo-operation but that never acquires a value will ultimately result in an error at load time.

To allow the assembler to calculate the sizes of functions for other tools, each `def/endef` pair that defines a function name must be matched by a `def/endef` pair after the function in which a storage class of -1 is assigned, where `-1` is the physical end of a function.

The paragraphs following describe the attribute-assigning operations (*attrasgn* in the syntax diagram just discussed). Keep in mind that all these operations apply to the symbol *name* that appeared in the opening `def` pseudo-operation.

`val` *expr* — Assigns the value *expr* to name. The type of the expression *expr* determines with which section *name* is associated. If value is ~, the current location in the `text` section is used.

`scl` *expr* — Declares a storage class for *name*. The expression *expr* must yield an absolute value that corresponds to the C compiler's internal representation of a storage class. The special value `-1` designates the physical end of a function.

`type` *expr* — Declares the C language type of *name*. The expression *expr* must yield an absolute value that corresponds to the C compiler's internal representation of a basic or derived type.

`tag` *expr* — Associates *name* with the structure, enumeration, or union named *str* that must have already been declared with a `def/endef` pair.

`line` *expr* — Provides the line number of *name*, where *name* is a block symbol. The expression *expr* should yield an absolute value that represents a line number.

`size` *expr* — Gives a size for *name*. The expression *expr* must yield an absolute value. When *name* is a structure or an array with a predetermined extent, *expr* gives the size in bytes. For bitfields, the size is in bits.

`dim` *expr1,expr2,...*

Indicates that *name* is an array. Each of the expressions must yield an absolute value that provides the corresponding array dimension.

Switch table operation

The C compiler generates a compact set of instructions for the C language `switch` construct. For example,

```
        sub.l &1,%d0
        cmp.l %d0,&4
        bhi   L%21
        add.w %d0,%d0
        mov.w 10(%pc,%d0.w),%d0
        jmp   6(%pc,%d0.w)
        swbeg &5
L%22:
        short L%15-L%22
        short L%21-L%22
        short L%16-L%22
        short L%21-L%22
        short L%17-L%22
```

The special `swbeg` pseudo-operation communicates to the assembler that the lines following it contain *rel* - *rel* subtractions. Remember that ordinarily such subtractions are risky because of span-dependent optimization. In this case, however, the assembler makes special allowances for the subtraction, because the compiler guarantees that both symbols will be defined in the current assembler file, and that one of the symbols is a fixed distance away from the current location.

The `swbeg` pseudo-operation takes an argument that looks like an immediate operand. The argument is the number of lines following the `swbeg` instruction that contain switch table entries. `swbeg` inserts two words into text. The first is the `illegal` instruction code. The second is the number of table entries that follow. The disassembler `dis`(1) needs the `illegal` instruction as a hint that what follows is a switch table. Otherwise, `dis`(1) becomes confused and tries to decode the table entries, which are differences between two symbols, as instructions.

Span-dependent optimization

The assembler makes certain choices about the object code it generates based on the distance between an instruction and its operand(s). Several choices are available; distances can be expressed with 8-, 16-, or 32-bit displacements. These are called the short, long, and very long forms, respectively.

Choosing the smallest, fastest form is called **span-dependent optimization.** Span-dependent optimization occurs most obviously in the choice of object code for branches and jumps. It also occurs when an operand may be represented by the program counter relative address mode instead of as an absolute two-word (`long`) address. The span-dependent optimization capability is normally enabled; the `-n` command option disables it. When this capability is disabled, the assembler makes worst case assumptions about the types of object code that must be generated. Span-dependent optimizations are performed only within `text` segment 0. Any reference outside `text` segment 0 is assumed to be a worst case.

The C compiler `c89`(1) generates branch instructions without a specific offset size. When the optimizer is used, it identifies branches that could be represented by the short form, and it changes the operation accordingly. The assembler chooses only between long and very long representations for branches.

Although the largest offset specification allowed is a word, large programs conceivably could have need for a branch to a location not reachable by a word displacement. Therefore, equivalent long forms of these instructions might be needed. When the assembler encounters a branch instruction without a size specification, it tries to choose between the long and very long forms of the instruction. If the operand can be represented in a word, the long form of the instruction will be generated. Otherwise, the very long form will be generated. For unconditional branches (for example, `br`, `bra`, and `bsr`), the very long form is just the equivalent jump (`jmp` and `jsr`) with an absolute (instead of `pc`-relative) address operand. For conditional branches, the equivalent very long form is a conditional branch around a jump, where the conditional test has been reversed.

Table 6-5 summarizes span-dependent optimizations. Again, the assembler chooses only between the long form and the very long form, while the optimizer chooses between the short and long forms for branches (but not `bsr`).

■ **Table 6-5** Assembler span-dependent optimizations

Instruction	Short form	Long form	Very long form
`br`, `bra`, `bsr`	Byte offset	Word offset	`jmp` or `jsr` with absolute long address
Conditional branch	Byte offset	Word offset	Short conditional branch with reversed condition around `jmp` with absolute long address
`jmp`, `jsr`		`pc`-relative address	Absolute long address
`lea`, `pea`		`pc`-relative address	Absolute long address

Branch instructions may have either a byte, word, or long `pc`-relative address operand. The assembler still chooses between word and long representations for branches if no byte size specification is given; however, the long form is replaced by a branch long with `pc`-relative address instead of a jump with absolute long address.

Address modes

The `as` assembler provides you with nine basic kinds of addressing modes:

- register direct
- register indirect
- register indirect with index
- memory indirect
- program counter indirect with displacement
- program counter indirect with index
- program counter memory indirect
- absolute
- immediate

Address mode syntax

In the tables throughout this chapter, the following abbreviations are used:

`A`*n*`/a`*n* Address register, where *n* is any digit from 0 through 7.

bd	2's-complement base displacement that is added before indirection takes place; the size may be 16 or 32 bits.
d	2's-complement or sign-extended displacement that is added as part of effective address calculation; size may be 8 or 16 bits; when omitted, the assembler uses the 0 value.
`Dn/dn`	Data register, where *n* is any digit from 0 through 7.
od	Outer displacement that is added as part of effective address calculation after memory indirection; the size may be 16 or 32 bits.
`PC/pc`	Program counter.
Ri/ri	Index register *i* may be any address or data register with an optional size designation (that is, *ri*`.w` for 16 bits or *ri*`.l` for 32 bits); the default size is 16 bits (`.w`).
scl	Optional scale factor that may be multiplied times index register in some modes. Values for *scl* are 1, 2, 4, or 8; the default is 1.
`[ ]`	Grouping characters used to enclose an indirect expression; these are required characters. Addressing arguments may occur in any order within the brackets.
`( )`	Grouping characters used to enclose an entire effective address; these are required characters. Addressing arguments may occur in any order within the parentheses.
`{ }`	Indicate that a scale factor is optional; these are not required characters.

It is important to note that expressions used for the absolute addressing modes need not be absolute expressions in the sense previously described in the section "Types." Although the addresses used in those addressing modes must ultimately be filled in with constants, that can be done later by the loader. The assembler need not be able to compute them. Indeed, the absolute long addressing mode is commonly used for accessing undefined external addresses. Several examples illustrate the use of this notation:

`%d7` indicates data register 7.

`(%a3)` indicates the contents of address register 3.

`14(%a4)` indicates that the decimal displacement 14 is added to the contents of address register 4.

`(%a1.w)` indicates the contents of the word in address register 1; the register is treated as an index register.

`(%d3.w{*2})` indicates the contents of the word in index register `d3`; the register is treated as an index register and the contents of the register are multiplied by 2.

Effective address modes

Table 6-6 summarizes the `as` syntax for MC68020 addressing modes.

■ **Table 6-6** Effective address modes

MC680x0 notation	as notation	Address mode
D*n*	%d*n*	Data register direct
A*n*	%a*n*	Address register direct
(A*n*)	(%a*n*)	Address register indirect
(A*n*)+	(%a*n*)+	Address register indirect with postincrement [1]
-(A*n*)	-(%a*n*)	Address register indirect with predecrement [1]
d(A*n*)	*d*(%a*n*)	Address register indirect with displacement (*d* signifies a signed 16-bit absolute displacement)
(A*n*, *Ri*)	(%a*n*, %*ri*.w) (%a*n*, %*ri*.l)	Address register indirect with index
d(A*n*, Ri)	*d*(%a*n*, %*ri*.w) *d*(%a*n*, %*ri*.l)	Address register indirect with index plus displacement (*d* signifies a signed 8-bit absolute displacement)
(A*n*, *Ri*{**scl*})	(%a*n*, %*ri*{**scl*})	Address register direct with index

(Continued)

■ **Table 6-6** Effective address modes (Continued)

MC680x0 notation	as notation	Address mode
(*bd*, A*n*, *Ri*{**scl*})	(*bd*, %a*n*, %*ri*{**scl*})	Address register direct with index plus base displacement

[1] If the address register is the stack pointer and the operand size is byte, the address is changed by 2 rather than 1 to keep the stack pointer aligned to a word boundary.

([*bd*,An,*Ri*{*scl*}],*od*)	([*bd*,%a*n*,%*ri*{*scl*}],*od*)	Memory indirect with preindexing plus base and outer displacement
([*bd*,An],*Ri*{*scl*},*od*)	([*bd*,%a*n*],%*ri*{*scl*},*od*)	Memory indirect with postindexing plus base and outer displacement
d(PC)	*d*(%pc)	Program counter indirect with displacement (*d* signifies 16-bit displacement)
d(PC,*Ri*)	*d*(%pc,%*rn*.l) *d*(%pc,%*rn*.w)	Program counter direct with index and displacement (*d* signifies 8-bit displacement)
(*bd*,PC,*Ri*{*scl*})	(*bd*,%pc,%*ri*{*scl*})	Program counter direct with index and base displacement
([*bd*,PC],*Ri*{*scl*},*od*)	([*bd*,%pc],%*ri*{*scl*},*od*)	Program counter memory indirect with postindexing plus base and outer displacement
([*bd*,PC,*Ri*{*scl*}],*od*)	([*bd*,%pc,%*ri*{*scl*}],*od*)	Program counter memory indirect with preindexing plus base and outer displacement
xxx.W	*xxx*	Absolute short address (*xxx* signifies an expression yielding a 16-bit memory address)
xxx.L	*xxx*	Absolute long address (*xxx* signifies an expression yielding a 32-bit memory address)
#*xxx*	&*xxx*	Immediate data (*xxx* signifies an absolute constant expression)

In Table 6-6, the index register notation should be understood as *ri*.*size***scale*, where both *size* and *scale* are optional. Section 2 of the *MC68020 32-Bit Microprocessor User's Manual* provides information about generating effective addresses and assembler syntax.

Note that suppressed address register %za*n* can be used in place of %a*n*, suppressed PC register %zpc may be used in place of %pc, and suppressed data register %zd*n* may be used in place of %d*n*, if suppression is desired.

Address modes for the MC68020 use two different formats of extension. The brief format provides fast indexed addressing, while the full format provides a number of options in size of displacement and indirection. The assembler will generate the brief format if the effective address expression is not memory indirect, value of displacement is within a byte, and no base or index suppression is specified; otherwise, the assembler will generate the full format.

Some variations of the MC68020 addressing modes may be redundant with the MC68000 address register indirect, address register indirect with displacement, and program counter with displacement modes. The assembler will select the more efficient mode when redundancy occurs. For example, when the assembler sees the form `(An)`, it will generate address register indirect mode.

The assembler will generate address register indirect with displacement when it encounters any of the following forms (as long as *bd* fits in 16 bits or less):

bd`(An)`

`(`*bd*`,An)`

`(An,` *bd*`)`

Machine instructions

The general forms of an MC68020 processor instruction are

inst

inst operand

inst operand, operand

where *inst* is the instruction followed by 0, 1, or 2 operands. An operand can be actual data (called an immediate operand) but often *operand* is the effective address of the data to be used in the instruction.

Table 6-7 shows how MC68020 instructions should be written to ensure the `as` assembler correctly understands them. Several abbreviations are used in Table 6-7:

A	The letter *A*, as in `add.`*A*, stands for one of the address operation size attribute letters `w` or `l`, representing a word or long operation, respectively.

CC

In the contexts `b`*CC*, `db`*CC*, and `s`*CC*, the letters *CC* represent any of the following condition code designations (except that `f` and `t` may not be used in the `b`*CC* instruction):

`cc`	Carry clear
`cs`	Carry set
`eq`	Equal
`f`	False
`ge`	Greater or equal
`gt`	Greater than
`hi`	High
`hs`	High or same (=`cc`)
`le`	Less or equal
`lo`	Low (=`cs`)
`ls`	Low or same
`lt`	Less than
`mi`	Minus
`ne`	Not equal
`pl`	Plus
`t`	True
`vc`	Overflow clear
`vs`	Overflow set

d

2's-complement or sign-extended displacement that is added as part of effective address calculation; the size may be 8 or 16 bits; when omitted, assembler uses value of 0.

EA

An arbitrary effective address.

(*eq*)

The two forms of machine instruction are equivalent.

FC

A function code that can be a data register, %sfc, %dfc, or an absolute expression with value 0-7 for MC68030 addressing or 0-15 for 68851 addressing.

I

An absolute expression representing a level, 0-7.

I

An absolute expression, used as an immediate operand.

L

A label reference, or any expression representing a memory address in the current segment.

mask

An absolute expression with value 0-7 for MC68030 addressing or 0-15 for 68851 addressing.

offset Either an immediate operand or a data register.

PCC One of the MC68851 PMMU condition codes.

Q An absolute expression evaluating to a number from 1 to 8.

S The letter *S*, as in `add.`*S*, stands for one of the operation size attribute letters `b`, `w`, or `l`, representing a byte, word, or long operation, respectively.

width Either an immediate operand or a data register.

Registers are designated using the following components:

`%` Register call.

`a` Address register.

`d` Data register.

r Either data or address register.

x, y, m, n Any digit from 0 through 7, where $x \neq y$, $m \neq n$, and $x \neq m$, and $y \neq n$.

These components are combined to form the following register designations:

`%a`*x*, `%a`*y*, `%a`*n* Address registers.

`%d`*x*, `%d`*y*, `%d`*n* Data registers.

`%`*mr* (P)MMU register (`%crp`, `%srp`, `%tt0`, `%tt1`, `%drp`, `%pcsr`, `%psr`, `%cal`, `%val`, `%scc`, `%ac`, `%bad`*x*, `%bac`*x*).

`%`*rc* Control register (`%sfc`, `%dfc`, `%cacr`, `%vbr`, `%caar`, `%msp`, `%isp`).

`%`*rx*, `%`*ry*, `%`*rn* Either data or address registers.

(*eq*) The two forms of machine instruction are equivalent.

■ **Table 6-7** MC68020 instruction formats

Mnemonic	Assembler syntax		Operation
ABCD	`abcd.b`	`%d`*y*`,%d`*x*	Add decimal with extend.
	`abcd.b`	`-(%a`*y*`),-(%a`*x*`)`	

(Continued)

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
ADD	`add.`*S*	*EA*,`%d`*n*	Add binary.
	`add.`*S*	`%d`*n*, *EA*	
ADDA	`add.`*A*	*EA*, `%a`*n*	Add address.
ADDI	`add.`*S*	`&`*I*,*EA*	Add immediate.
ADDQ	`add.`*S*	`&`*Q*,*EA*	Add quick.
ADDX	`addx.`*S*	`%d`*y*, `%d`*x*	Add extended.
	`addx.`*S*	`-(%ay),-(%ax)`	
AND	`and.`*S*	*EA*, `%d`*n*	AND logical.
	`and.`*S*	`%d`*n*, *EA*	
ANDI	`and.`*S*	`&`*I*, *EA*	AND immediate.
ANDI to CCR	`and.b`	`&`*I*, `%cc`	AND immediate to condition code register.
ANDI to SR	`and.w`	`&`*I*, `%sr`	AND immediate to the status register.
ASL	`asl.`*S*	`%d`*x*, `%d`*y*	Arithmetic shift (left).
	`asl.`*S*	`&`*Q*, `%d`*y*	
	`asl.w`	`&`1,*EA*	
ASR	`asr.`*S*	`%d`*x*, `%d`*y*	Arithmetic shift (right).
	`asr.`*S*	`&`*Q*, `%d`*y*	
	`asr.w`	`&1`, *EA*	
Bcc	`b`*CC*	*L*	Branch conditionally (16-bit displacement).
	`b`*CC*`.b`	*L*	Branch conditionally (short) (8-bit displacement).
	`b`*CC*`.l`	*L*	Branch conditionally (long) (32-bit displacement).
BCHG	`bchg`	`%d`*n*, *EA*	Test a bit and change .
	`bchg`	`&I`, *EA*	Note: `bchg` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is used.
BCLR	`bclr`	`%d`*n*,*EA*	Test a bit and clear.
	`bclr`	*&I*,*EA*	Note: `bclr` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is used.
BFCHG	`bfchg`	*EA*{*offset*:*width*}	Complement bitfield.
BFCLR	`bfclr`	*EA*{*offset*:*width*	Clear bitfield.

(Continued)

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
BFEXTS	`bfexts`	*EA*{*offset*:*width*}, `%d`*n*	Extract bitfield (signed).
BFEXTU	`bfextu`	*EA*{*offset*:*width*}, `%d`*n*	Extract bitfield (unsigned).
BFFFO	`bfffo`	*EA*{*offset*:*width*}, `%d`*n*	Find first one in bitfield.
BFINS	`bfins`	`%d`*n*, *EA*{*offset*:*width*}	Insert bitfield.
BFSET	`bfset`	*EA*{*offset*:*width*}	Set bitfield.
BFTST	`bftst`	*EA*{*offset*:*width*}	Test bitfield.
BKPT	`bkpt`	`&`*I*	Breakpoint.
BRA	`bra.`*S*	*L*	Branch always.
	`br.`*S*	*L*	Same as `bra.`*S*.
BSET	`bset`	`%d`*n*,*EA*	Test a bit and set.
	`bset`	`&`*I*, *EA*	Note: `bset` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is used.
BSR	`bsr.`*S*	*L*	Branch to subroutine.
BTST	`btst`	`%d`*n*,*EA*	Test a bit and set.
	`btst`	`&`*I*, *EA*	Note: `btst` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is used.
CALLM	`callm`	`&`*I*,*EA*	Call module.
CAS	`cas.`*S*	`%d`*x*, `%d`*y*, *EA*	Compare and swap operands.
CAS2	`cas2.`*S*	`%d`*x*:`%d`*y*, `%d`*m*:`%d`*n*, (`%`*rx*):(`%`*ry*)	Compare and swap dual operands.
CHK	`chk.`*A*	*EA*, `%d`*n*	Check register against bounds.
CHK2	`chk2.`*S*	*EA*, `%`*rn*	Check register against bounds.
CLR	`clr.`*S*	*EA*	Clear an operand.
CMP	`cmp.`*S*	`%d`*n*, *EA*	Compare. [2]
CMPA	`cmpa.`*A*	`%a`*n*, *EA*	Compare address [2,3]
CMPI	`cmpi.`*S*	*EA*, `&`*I*	Compare immediate. [2,3]
CMPM	`cmpm.`*S*	(`%a`*x*)+, (`%a`*y*)+	Compare memory. [2,3]
CMP2	`cmp2.`*S*	`%`*rn*, *EA*	Compare register against bounds. [3]

(Continued)

[2]The order of operands in `as` is the reverse of that in the *MC68000 Programmer's Reference Manual.*

[3]The `cmp.`*S* syntax is also recognized.

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
DBcc	db*CC*	%d*n*, *L*	Test condition, decrement, and branch
	dbra	%d*n*, *L*	Decrement and branch always.
DIVS	divs.w	*EA*, %d*x*	Signed divide 32/16 -> 16r:16q.
	tdivs.l	*EA*, %d*x*	Signed divide (long) 32/32 -> 32q.
	divs.l	*EA*, %d*x*	Signed divide (long) 32/32 -> 32r:32q. [4]
	divs.l	*EA*, %d*x*:%d*y*	
DIVSL	tdivs.l	*EA*, %d*x*:%d*y*	Signed divide (long) 64/32 -> 32r:32q.
DIVU	divu.w	*EA*, %d*n*	Unsigned divide 32/16 -> 16r:16q.
	tdivu.l	*EA*, %d*x*	Unsigned divide (long) 32/32 -> 32*(eq)*.
	divu.l	*EA*, %d*x*	
DIVUL	divu.l	*EA*, %d*x*:%d*y*	Unsigned divide (long) 64/32 -> 32r:32q. [5]
	tdivu.l	*EA*, %d*x*:%d*y*	Unsigned divide (long) 32/32 -> 32r:32q. [6]
EOR	eor.*S*	%d*n*, *EA*	Exclusive OR logical
EORI	eor.*S*	&*I*, *EA*	Exclusive OR immediate.
EORI to CCR	eor.b	&*I*, %cc	Exclusive OR immediate to condition code register.
EORI to SR	eor.w	&*I*, %sr	Exclusive OR immediate to the status register.
EXG	exg	%r*x*, %r*y*	Exchange registers.
EXT	ext.w	%d*n*	Sign-extend low-order byte of data to word.
	ext.l	%d*n*	Sign-extend low-order word of data to long.
EXTB	extw.l	%d*n*	Same as ext.l.
	extb.l	%d*n*	Sign-extend low-order byte of data to long.

[4] Whenever %d*x* and %d*y* are the same register, then the form is equivalent to the divs.l *EA*, %d*x* form.

[5] Whenever %d*x* and %d*y* are the same register, then the form is equivalent to the divu.l *EA*, %d*x* form

[6] Whenever %d*x* and %d*y* are the same register, then the form is equivalent to the tdivu.l *EA*, %d*x* form.

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
ILLEGAL	`illegal`		Illegal instruction.
JMP	`jmp`	*EA*	Jump.
JSR	`jsr`	*EA*	Jump to subroutine.
LEA	`lea`	*EA*, `%a`*n*	Load effective address.
LINK	`link.`*A*	`%a`*n*, `&`*I*	Link and allocate.
LSL	`lsl.`*S*	`%d`*x*, `%d`*y*	Logical shift (left).
	`lsl.`*S*	`&`*Q*, `%d`*y*	
	`lsl.`*S*	*EA*	
LSR	`lsr.`*S*	`%d`*x*, `%d`*y*	Logical shift (right).
	`lsr.`*S*	`&`*Q*, `&d`*y*	
	`lsr.`*S*	*EA*	
MOVE	`move.`*S*	*EA*, *EA*	Move data from source to destination. [7,8]
MOVE to CCR	`move.w`	*EA*, `%cc`	Move to condition code register. [7]
MOVE from CCR	`move.w`	`%cc`, *EA*	Move from condition code register. [7]
MOVE to SR	`move.w`	*EA*, `%sr`	Move to the status register. [7]
MOVE from SR	`move.w`	`%sr`, *EA*	Move from the status register. [7]
MOVE USP	`move.l`	`%usp`, `%a`*n*	Move user stack pointer. [7]
	`move.l`	`%a`*n*, `%usp`	
MOVEA	`move.`*A*	*EA*, `%a`*n*	Move address. [7]
MOVEC	`move.l`	`%`*rc*, `%`*rn*	Move from/to control register. [7]
	`move.l`	`%`*rn*, `%`*rc*	
MOVEM	`movem.`*A*	*EA*, `&`*I*	Move multiple registers. [7,9]
	`movem.`*A*	`&`*I*, *EA*	
MOVEP	`movep.`*A*	`%d`*x*, *d*`(%a`*y*`)`	Move peripheral data. [7]
	`movep.`*A*	*d*`(%a`*y*`)`, `%d`*x*	
MOVEQ	`move.l`	`&`*I*, `%d`*n*	Move quick. [7]

[7]In all move commands, `move` may be shortened to `mov`.

[8]If the destination is an address register, the instruction generated is `MOVEA`.

[9]The immediate operand is a mask designating which registers are to be moved to memory or which are to receive memory data. Not all addressing modes are permitted, and the correspondence between mask bits and register numbers depends on the addressing mode.

MOVES	`moves.`*S*	`%`*rn*, *EA*	Move to/from address space. [7]
	`moves.`*S*	*EA*, `%`*rn*	

(Continued)

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
MULS	`muls.w`	*EA*, `%d`*x*	Signed multiply 16*16 -> 32.
	`tmuls.l`	*EA*, `%d`*x*	Signed multiply (long) 32*32 -> 32 (*eq*).
	`muls.l`	*EA*, `%d`*x*	
	`muls.l`	*EA*, `%d`*x*:`%d`*y*	Signed multiply (long) 32*32 -> 64.
MULU	`mulu.w`	*EA*, `%d`*x*	Unsigned multiply 16*16 -> 32.
	`tmulu.l`	*EA*, `%d`*x*	Unsigned multiply (long) 32*32 -> 32*(eq)*.
	`mulu.l`	*EA*, `%d`*x*	
	`mulu.l`	*EA*, `%d`*x*:`%d`*y*	Unsigned multiply (long) 32*32 -> 64.
NBCD	`nbcd`	*EA*	Negate decimal with extend.
NEG	`neg.`*S*	*EA*	Negate.
NEGX	`negx.`*S*	*EA*	Negate with extend.
NOP	`nop`		No operation.
NOT	`not.`*S*	*EA*	Logical complement.
OR	`or.`*S*	*EA*, `%d`*n*	Inclusive OR logical.
	`or.`*S*	`%d`*n*, *EA*	
ORI	`ori.`*S*	`&`*I*, *EA*	Inclusive OR immediate. Equivalent to `or.`*S*.
ORI to CCR	`ori.w`	`&`*I*, `%cc`	Inclusive OR immediate to condition code register. Equivalent to `or.w`.
ORI to SR	`ori.w`	`&`*I*, `%sr`	Inclusive OR immediate to the status register. Equivalent to `or.w`.
PACK	`pack`	`-(%a`*x*`),-(%a`*y*`),&`*I*	Pack BCD.
	`pack`	`%d`*x*, `%d`*y*, `&`*I*	

PFLUSH	`pflush`	*FC, &mask*	Invalidate set of ATC.entries with the given function code [10]

(Continued)

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
PFLUSH	`pflush`	*FC, &mask, EA*	Invalidate set of ATC.entries with the given function code and effective address. [10]
PFLUSHS	`pflushs`	*FC, &mask*	Invalidate set of ATC.entries with the given function code, even if SGS bit is set. [11]
	`pflushs`	*FC, &mask, EA*	Invalidate set of ATC.entries with the given function code and effective address, even if SGS bit is set. [10]
PFLUSHA	`pflusha`		Invalidate all entries in the ATC.[10]
PFLUSHR	`pflushr`	*EA*	Invalidate ATC and RPT entries matching effective address.[11]
PLOADR	`ploadr`	*FC, EA*	Load an entry into the ATC for read access. [10]
PLOADW	`ploadw`	*FC, EA*	Load an entry into the ATC for write access. [10]
PMOVE	`pmove`	*%mr, EA*	Move data from MMU register to destination. [10]
	`pmove`	*EA, %mr*	Move data from destination to MMU register. [10]
PMOVEFD	`pmove`	*EA, %mr*	Move data from destination to MMU register. [12]
PRESTORE	`prestore`	*EA*	Restore function. [11]
PSAVE	`psave`	*EA*	Save function. [11]
PTESTR	`ptestr`	*FC, EA, &I*	Get information about logical address; set bit for read. [10]

[10]These instructions are available on MC68030 and MC68851 only; additional MC68851 instructions are shown in Table 6-14.

[11]These instructions are available on the MC68851 only.

[12]This instruction is available on the MC68030 only.

	`ptestr`	*FC, EA,* `&`*I,* `%a`*x*	Get information about logical address and load register; set bit for read. [10]
PTESTW	`ptestw`	*FC, EA,* `&`*I*	Get information about logical address; set bit for write. [10]
	`ptestw`	*FC, EA,* `&`*I,* `%a`*x*	Get information about logical address and load register; set bit for write. [10]

(Continued)

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
PTRAPpcc	`ptrap`	*PCC*	Trap on PMMU condition. [11]
PVALID	`pvalid`	`%`*val, EA*	Validate a pointer against VAL register. [11]
	`pvalid`	`%a`*x, EA*	Validate a pointer against address register. [11]
PEA	`pea`	*EA*	Push effective address.
RESET	`reset`		Reset external devices.
ROL	`rol.`*S*	`%d`*x*, `%d`*y*	Rotate left without extend.
	`rol.`*S*	`&`*Q*, `%d`*y*	
	`rol.w`	*EA*	
ROR	`ror.`*S*	`%d`*x*, `%d`*y*	Rotate right without extend)
	`ror.`*S*	`&`*Q*, `%d`*y*	
	`ror.w`	*EA*	
ROXL	`roxl.`*S*	`%d`*x*, `%d`*y*	Rotate left with extend.
	`roxl.`*S*	`&`*Q*, `%d`*y*	
	`roxl.w`	*EA*	
ROXR	`roxr.`*S*	`%d`*x*, `%d`*y*	Rotate right with extend.
	`roxr.`*S*	`&`*Q*, `%d`*y*	
	`roxr.w`	*EA*	
RTD	`rtd`	`&`*I*	Return and deallocate parameters.
RTE	`rte`	`%`*rn*	Return from exception.
RTM	`rtm`		Return from module.
RTR	`rtr`		Return and restore condition codes.
RTS	`rts`		Return from subroutine.
SBCD	`sbcd`	`%d`*y*, `%d`*x*	Subtract decimal with extend.
	`sbcd`	`-(%a`*y*`), -(%a`*x*`)`	

Scc	s*CC*	*EA*	Set according to condition.
STOP	`stop`	`&`*I*	Load status register and stop.
SUB	`sub.`*S*	`EA,%d`*n* `%d`*n*`, EA`	Subtract binary.
SUBA	`sub.`*A*	`EA,%a`*n*	Subtract address.
SUBI	`sub.`*S*	`&`*I*`, EA`	Subtract immediate (`subi` also works).
SUBQ	`sub.`*S*	`&`*Q*`, EA`	Subtract quick (`subq` also works).

(Continued)

■ **Table 6-7** MC68020 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
SUBX	`subx.`*S*	`%d`*y*`,%d`*x* `-(%a`*y*`), -(%a`*x*`)`	Subtract with extend.
SWAP	`swap`	`%d`*n*	Swap register halves.
TAS	`tas`	*EA*	Test and set an operand.
TRAP	`trap`	`&`*I*	Trap.
TRAPV	`trapv`		Trap on overflow.
TRAPcc	`t`*CC* `trap`*CC* `tp`*CC*`.`*A* `trap`*CC*`.`*A*	 `&`*I* `&`*I*	Trap on condition (*eq*). (*eq*)
TST	`tst.`*S*	*EA*	Test an operand.
UNLK	`unlk`	`%a`*n*	Unlink.
UNPK	`unpk` `unpk`	`-(%a`*x*`), -(%a`*y*`),&`*I* `%d`*x*`,%d`*y*`,&`*I*	Unpack BCD.

Instructions for the MC68881

All A/UX-capable systems are equipped with floating-point coprocessor capability (on some systems this is embedded in the MC68030 processor). The coprocessor uses special instructions, the understanding of which requires an introduction to concepts new to programmers who have not used such coprocessors before.

A/UX and the MC68881 coprocessor fully support the IEEE standard for handling NaN (Not a Number) conditions. To maximize flexibility there are two modes of handling unordered conditions: non-IEEE and IEEE. These condition codes are shown in Tables 6-8 and 6-9 respectively. Non-IEEE conditions codes are used in two cases:

- when porting a program that does not support the IEEE standard
- when generating code that does not support the IEEE floating-point concepts (for example, the unordered condition)

When non-IEEE condition codes are used, an exception is generated if an unordered condition is found. It is the responsibility of the application to test for and handle these conditions. Generally, A/UX users will want to use the IEEE condition codes.

- **Table 6-8** Non-IEEE condition codes

CC	Meaning
`ge`	Greater than or equal
`gl`	Greater or less than
`gle`	Greater or less than or equal
`gt`	Greater than
`le`	Less than or equal
`lt`	Less than
`ngt`	Not greater than
`nge`	Not (greater than or equal)
`nlt`	Not less than
`ngl`	Not (greater or less than)
`nle`	Not (less than or equal)
`ngle`	Not (greater or less than or equal)
`sneq`	Signaling not equal
`sf`	Signaling false
`seq`	Signaling equal
`st`	Signaling true

- **Table 6-9** IEEE condition codes

CC	Meaning
`eq`	Equal
`oge`	Ordered greater than or equal
`ogl`	Ordered greater or less than
`ogt`	Ordered greater than
`ole`	Ordered less than or equal
`olt`	Ordered less than

`or`	Ordered
`t`	True
`ule`	Unordered or less or equal
`ult`	Unordered or less than
`uge`	Unordered or greater than or equal
`ueq`	Unordered or equal
`ugt`	Unordered or greater than
`un`	Unordered
`neq`	Not equal
`f`	False

The floating-point coprocessor also supports a set of standard constants that are kept in ROM. The MC68881 constant ROM values are shown in Table 6-10. In this table, *ccc* indicates a constant condition code designator.

■ **Table 6-10** Constants in MC68881 ROM

ccc	Value	ccc	Value
`0x0`	pi	`3x5`	10**4
`0xB`	log10(2)	`3x6`	10**8
`0xC`	e	`3x7`	10**16
`0xD`	log2(e)	`3x8`	10**32
`0xE`	log10(e)	`3x9`	10**64
`0xF`	0.0	`3xA`	10**128
`3x0`	ln(2)	`3xB`	10**256
`3x1`	ln(10)	`3xC`	10**512
`3x2`	10**0	`3xD`	10**1024
`3x3`	10**1	`3xE`	10**2048
`3x4`	10**2	`3xF`	10**4096

Table 6-11 shows how the floating-point coprocessor (MC68881) instructions should be written to be understood by the `as` assembler. Abbreviations used in Table 6-11 are

A	Source format letters `w` or `l`
B	Source format letters `b`, `w`, `l`, `s`, or `p`
CC	Any of the floating-point condition code designations listed in Table 6-8 and Table 6-9.

ccc Any of the ROM constants listed in Table 6-10.

EA An effective address

I An absolute expression, used as an immediate operand

L A label reference or any expression representing a memory address in the current segment

SF Source format letters:
`b`=byte integer
`d`=double precision
`l`=long word integer
`p`=packed binary code decimal
`s`=single precision
`w`=word integer
`x`=extended precision

- *Note:* The source format must be specified if more than one source format is permitted, otherwise a default source format of extended precision (`x`) is assumed. Source format need not be specified if only one format is permitted by the operation.

`%`*control* Floating-point control register

`%d`*n* Data register, where $0 \leq n \leq 7$

`%fpcr` Floating-point control register

`%fpiar` Floating-point instruction address register

`%fp`*m*`,%fp`*n*`,%fp`*q* Floating-point data registers, where *m*, *n*, and *q* are digits from 0 through 7

`%fpsr` Floating-point status register

`%`*iaddr* Floating-point instruction address register

`%`*status* Floating-point status register

■ **Table 6-11** MC68881 instruction formats

Mnemonic	Assembler syntax		Operation
FABS	`fabs.`*SF*	*EA*, `%fp`*n*	Absolute value function.
	`fabs.x`	`%fp`*m*, `%fp`*n*	
	`fabs.x`	`%fp`*n*	
FACOS	`facos.`*SF*	*EA*, `%fp`*n*	Arccosine function.
	`facos.x`	`%fp`*m*, `%fp`*n*	
	`facos.x`	`%fp`*n*	
FADD	`fadd.`*SF*	*EA*, `%fp`*n*	Floating-point add.
	`fadd.x`	`%fp`*m*, `%fp`*n*	
FASIN	`fasin.`*SF*	*EA*, `%fp`*n*	Arcsine function.
	`fasin.x`	`%fp`*m*, `%fp`*n*	
	`fasin.x`	`%fp`*n*	
FATAN	`fatan.`*SF*	*EA*, `%fp`*n*	Arctangent function.
	`fatan.x`	`%fp`*m*, `%fp`*n*	
	`fatan.x`	`%fp`*n*	
FATANH	`fatanh.`*SF*	*EA*, `%fp`*n*	Hyperbolic arctangent function.
	`fatanh.x`	`%fp`*m*, `%fp`*n*	
	`fatanh.x`	`%fp`*n*	
FBcc	`fb`*CC.A*	*L*	Coprocessor branch conditionally.
FCMP	`fcmp.`*SF*	`%fp`*n*, *EA*	Floating-point compare. [13]
	`fcmp.x`	`%fp`*n*, `%fp`*m*	
FCOS	`fcos.`*SF*	*EA*, `%fp`*n*	Cosine function.
	`fcos.x`	`%fp`*m*, `%fp`*n*	
	`fcos.x`	`%fp`*n*	
FCOSH	`fcosh.`*SF*	*EA*, `%fp`*n*	Hyperbolic cosine function.
	`fcosh.x`	`%fp`*m*, `%fp`*n*	
	`fcosh.x`	`%fp`*n*	

*The order of operands in `as` is the reverse of that in the *MC68881 Programmer's Reference Manual*.

(Continued)

■ **Table 6-11** MC68881 instruction formats (Continued)

[13]The order of operands in `as` is the reverse of that in the *MC68881 Programmer's Reference Manual*.

Mnemonic	Assembler syntax		Operation
FDBcc	`fdb`*CC*`.w`	`%d`*n*, *L*	Decrement and branch on condition.
FDIV	`fdiv.`*SF*	*EA*, `%fp`*n*	Floating-point divide.
	`fdiv.x`	`%fp`*m*, `%fp`*n*	
FETOX	`fetox.`*SF*	*EA*, `%fp`*n*	e^x function.
	`fetox.x`	`%fp`*m*, `%fp`*n*	
	`fetox.x`	`%fp`*n*	
FETOXM1	`fetoxml.`*SF*	*EA*, `%fp`*n*	$e^{x(x-1)}$ function.
	`fetoxml.x`	`%fp`*m*, `%fp`*n*	
	`fetoxml.x`	`%fp`*n*	
FGETEXP	`fgetexp.`*SF*	*EA*, `%fp`*n*	Get the exponent function.
	`fgetexp.x`	`%fp`*m*, `%fp`*n*	
	`fgetexp.x`	`%fp`*n*	
FGETMAN	`fgetman.`*SF*	*EA*, `%fp`*n*	Get the mantissa function.
	`fgetman.x`	`%fp`*m*, `%fp`*n*	
	`fgetman.x`	`%fp`*n*	
FINT	`fint.`*SF*	*EA*, `%fp`*n*	Integer part function.
	`fint.x`	`%fp`*m*, `%fp`*n*	
	`fint.x`	`%fp`*n*	
FINTRZ	`fintrz.`*SF*	*EA*, `%fp`*n*	Integer part, round-to-zero function.
	`fintrz.x`	`%fp`*m*, `%fp`*n*	
	`fintrz.x`	`%fp`*n*	
FLOG2	`flog2.`*SF*	*EA*, `%fp`*n*	Binary log function.
	`flog2.x`	`%fp`*m*, `%fp`*n*	
	`flog2.x`	`%fp`*n*	
FLOG10	`flog10.`*SF*	*EA*, `%fp`*n*	Common log function.
	`flog10.x`	`%fp`*m*,`%fp`*n*	
	`flog10.x`	`%fp`*n*	

(Continued)

■ **Table 6-11** MC68881 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
FLOGN	`flogn.`*SF*	*EA*, `%fp`*n*	Natural log function.
	`flogn.x`	`%fp`*m*, `%fp`*n*	
	`flogn.x`	`%fp`*n*	
FLOGNP1	`flognp1.`*SF*	*EA*, `%fp`*n*	Natural log (x+1) function.
	`flognp1.x`	`%fp`*m*, `%fp`*n*	
	`flognp1.x`	`%fp`*n*	

FMOD	`fmod.`*SF*	*EA*, `%fp`*n*	Floating point modulo.
	`fmod.x`	`%fp`*m*, `%fp`*n*	
FMOVE	`mov.`*SF*	*EA*, `%fp`*n*	Move to floating-point register. [14]
	`fmov.x`	`%fp`*m*, `%fp`*n*	
	`fmove.`*SF*	`%fp`*n*, *EA*	Move from floating-point register to memory. [14]
	`fmove.p`	`%fp`*n*, *EA*`{&`*I*`}`	
	`fmove.p`	`%fp`*n*, *EA*`{%d`*n*`}`	
	`fmove.l`	*EA*, `%`*control*	Move from memory to special register. [14]
	`fmove.l`	*EA*, `%`*status*	
	`fmove.l`	*EA*, `%`*iaddr*	
	`fmove.l`	`%`*control*, *EA*	Move to memory from special register. [14]
	`fmove.l`	`%`*status*, *EA*	
	`fmove.l`	`%`*iaddr*, *EA*	

(Continued)

■ **Table 6-11** MC68881 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
FMOVECR	`fmovcr.x`	`&`*ccc*, `%fp`*n*	Move a ROM-stored value to a floating-point register. [14, 15, 16]
FMOVEM	`fmovem.x`	*EA*, `&`*I*	Move to multiple floating point register. [14, 15]
	`fmovem.x`	`&`*I*, *EA*	Move from multiple registers to memory. [14, 15]
	`fmovem.x`	*EA*, `%d`*n*	Move to a data register. [14]
	`fmovem.x`	`%d`*n*, *EA*	Move a data register to memory. [14]
	`fmovem.l`	`%`*control*, *EA*	Move to special registers (1, 2, or 3 registers, separated by commas). [14]
	`fmovem.l`	`%`*status*, *EA*	

[14] In all (floating-point) move commands, `move` may be shortened to `mov`.

[15] The immediate operand is a mask designating which registers are to be moved to memory or which registers are to receive memory data. Not all addressing modes are permitted and the correspondence between mask bits and register numbers depends on the addressing mode used.

[16] See Table 6-10, "Constants in MC68881 ROM."

	`fmovem.l`	*%iaddr, EA*	
	`fmovem.l`	*EA, %control*	Move from special registers (1, 2, or 3 registers, separated by commas). [14]
	`fmovem.l`	*EA, %status*	
	`fmovem.l`	*EA, %iaddr*	
FMUL	`fmul.`*SF*	*EA,* `%fp`*n*	Floating-point multiply.
	`fmul.x`	`%fp`*m,* `%fp`*n*	
FNEG	`fneg.`*SF*	*EA,* `%fp`*n*	Negate function.
	`fneg.x`	`%fp`*m,* `%fp`*n*	
	`fneg.x`	`%fp`*n*	

(Continued)

■ **Table 6-11** MC68881 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
FNOP	`fnop`		Floating-point no-op.
FREM	`frem.`*SF*	*EA,* `%fp`*n*	Floating-point remainder.
	`frem.x`	`%fp`*m,* `%fp`*n*	
FRESTORE	`frestore`	*EA*	Restore internal state of coprocessor.
FSAVE	`fsave`	*EA*	Coprocessor save.
FSCALE	`fscale.`*SF*	*EA,* `%fp`*n*	Floating-point scale exponent.
	`fscale.x`	`%fp`*m,* `%fp`*n*	
FScc	`fs`*CC*`.b`	*EA*	Set on condition.
FSGLDIV	`fsgldiv.`*B*	*EA,* `%fp`*n*	Floating-point single-precision divide.
	`fsgldiv.s`	`%fp`*m,* `%fp`*n*	
FSGLMUL	`fsglmul.`*B*	`fsglmul.s`	Floating-point single-precision multiply.
	`fsglmul.s`	`%fp`*m,* `%fp`*n*	
FSIN	`fsin.`*SF*	*EA,* `%fp`*n*	Sine function.
	`fsin.x`	`%fp`*m,* `%fp`*n*	
	`fsin.x`	`%fp`*n*	
FSINCOS	`fsincos.`*SF*	*EA,* `%fp`*n*`:%fp`*q*	Sine/cosine function.
	`fsincos.x`	`%fp`*m,* `%fp`*n*`:%fp`*q*	
FSINH	`fsinh.`*SF*	*EA,* `%fp`*n*	Hyperbolic sine function.
	`fsinh.x`	`%fp`*m,%fp*`n`	
	`fsinh.x`	`%fp`*n*	
FSQRT	`fsqrt.`*SF*	*EA,* `%fp`*n*	Square root function.
	`fsqrt.x`	`%fp`*m,* `%fp`*n*	

	`fsqrt.x`	`%fp`*n*	
FSUB	`fsub.`*SF*	*EA*, `%fp`*n*	Floating-point subtract.
	`fsub.x`	`%fp`*m*, `%fp`*n*	

(Continued)

■ **Table 6-11** MC68881 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
FTAN	`ftan.`*SF*	*EA*, `%fp`*n*	Tangent function.
	`ftan.x`	`%fp`*m*, `%fp`*n*	
	`ftan.x`	`%fp`*n*	
FTANH	`ftanh.`*SF*	*EA*, `%fp`*n*	Hyperbolic tangent function.
	`ftanh.x`	`%fp`*m*, `%fp`*n*	
	`ftanh.x`	`%fp`*n*	
FTENTOX	`ftentox.`*SF*	*EA*, `%fp`*n*	10^x function.
	`ftentox.x`	`%fp`*m*, `%fp`*n*	
	`ftentox.x`	`%fp`*n*	
FTcc	`ft`*CC*		Trap on condition without a parameter.
FTRAPcc	`ftrap`*CC*		Trap on condition without a parameter.
FTPcc	`ftp`*CC*`.`*A*	`&`*I*	Trap on condition with a parameter.
FTRAPcc	`ftrap`*CC*`.`*A*	`&`*I*	Trap on condition with a parameter.
FTST	`ftest.`*SF*	*EA*	Floating-point test an operand *Note:* The `ftst` form (floating-point trap on signal true) is no longer supported due to a conflict with the FTST (floating-point test an operand instruction).
	`ftest.x`	`%fp`*m*	
	`ftst.`*SF*	*EA*	
	`ftst.x`	`%fp`*m*	
FTWOTOX	`ftwotox.`*SF*	*EA*, `%fp`*n*	2^x function.
	`ftwotox.x`	`%fp`*m*, `%fp`*n*	
	`ftwotox.x`	`%fp`*n*	

Instructions for the MC68851

The tables in this section show how the paged memory management unit (PMMU) (MC68851) instructions should be written to be understood by the `as` assembler.

The conditions that the PMMU tests can be either set or cleared. Tables 6-12 and 6-13 show the mnemonics for these states. In Table 6-14, *CC* represents any of the condition code designations depicted in Tables 6-12 and 6-13.

■ **Table 6-12** PMMU condition codes: Condition is set

CC	Meaning
`bs`	Bus error
`ls`	Limit violation
`ss`	Supervisor violation
`as`	Access level violation
`ws`	Write protected
`is`	Invalid
`gs`	Gate
`cs`	Globally shared

■ **Table 6-13** PMMU condition codes: Condition is clear

CC	Meaning
`bc`	Bus error
`lc`	Limit violation
`sc`	Supervisor violation
`ac`	Access level violation
`wc`	Write protected
`ic`	Invalid
`gc`	Gate
`cc`	Globally shared

Additional abbreviations used in Table 6-14 are as follows:

D — Represents an absolute expression used as an immediate operand depth level in the `PTESTR/PTESTW` instructions, where $0 \leq D \leq 7$.

EA — Represents an effective address.

FC — Represents one of the following function codes:

I	An absolute expression used as an immediate operand.
`%dfc`	The destination function code register.
`%d`*n*	A data register.
`%sfc`	The source function code register.
`%sfcr`	The source function code register.

I Represents an absolute expression used as an immediate operand.

L A label reference or any expression representing a memory address in the current segment.

M Represents an absolute expression used as an immediate operand mask in the `PFLUSH/PFLUSHS` instructions, where $0 \leq M \leq 15$.

`%a`*n* Represents an address register 0 through 7.

`%d`*n* Represents a data register 0 through 7.

`%`*pm* Represents one of the following PMMU registers:

`%ac`	Access control register
`%bac`	Breakpoint acknowledge control register 0 through 7
`%bad`	Breakpoint acknowledge data register 0 through 7
`%cal`	Current access level register
`%crp`	CPU root pointer register
`%drp`	DMA root pointer register
`%pcsr`	Cache status register
`%psr`	Status register
`%scc`	Stack change control register
`%srp`	Supervisor root pointer register
`%tc`	Transition control register
`%val`	Validate access level register

- *Note:* The source format must be specified if more than one source format is permitted, otherwise a default source format of `w` is assumed. Source format need not be specified if only one format is permitted by the operation.

■ **Table 6-14** MC68851 instruction formats

Mnemonic	Assembler syntax		Operation
PB*cc*	`pb`*CC.A*	*L*	Branch on PMMU condition.
PDB*cc*	`pdb`*CC*`.w`	`%d`*n*, *L*	Test, decrement, branch.
PFLUSH	`pflush`	*FC*, `&`*M*	Invalidate entries in ATC.
	`pflush`	*FC*, `&`*M*, *EA*	
PFLUSHA	`pflusha`		Invalidate all ATC entries.
PFLUSHS	`pflushs`	*FC*, `&`*M*	Invalidate entries in ATC including shared entries.
	`pflushs`	*FC*, `&`*M*, *EA*	
PFLUSHR	`pflushr`	*EA*	Invalidate ATC and RPT entries.
PLOADR	`ploadr`	*FC*, *EA*	Load an entry into ATC.

(Continued)

■ **Table 6-14** MC68851 instruction formats (Continued)

Mnemonic	Assembler syntax		Operation
PLOADW	`ploadw`	*FC, EA*	Load an entry into ATC.
PMOVE	`pmove.`*A*	`%`*pm, EA*	Move PMMU register. [17]
	`pmove.`*A*	*EA,* `%`*pm*	
PRESTORE	`prestore`	*EA*	PMMU restore function.
PSAVE	`psave`	*EA*	PMMU save function.
PScc	`ps`*CC*	*EA*	Set on PMMU condition.
PTESTR	`ptestr`	*FC, EA,* `&`*D*	Get information about logical address.
	`ptestr`	*FC, EA,* `&`*D,* `%a`*n*	
PTESTW	`ptestw`	*FC, EA,* `&`*D*	Get information about logical address
	`ptestw`	*FC, EA,* `&`*D,* `%a`*n*	
PTRAPcc	`pt`*CC*		Trap on PMMU condition
	`ptrap`*CC*		
	`pt`*CC.A*		
	`ptrap`*CC.A*		
PVALID	`pvalid`	`%val,` *EA*	Validate a pointer
	`pvalid`	`%a`*n, EA*	

[17] The `pmov.` syntax is also recognized.

Chapter 7 The `ld` loader

This chapter describes the A/UX loader, `ld`, which creates executable object files by combining object files, performing relocation, and resolving external references. `ld` also processes symbolic debugging information. The input to `ld` is made up of relocatable object files produced by a compiler, an assembler, or a previous `ld` run. The loader combines these object files to form either a relocatable or an absolute (executable) object file. In other documentation the loader is also called a *linker* or *link editor*.

`ld` supports a command language that lets you control the loading process with great flexibility and precision. Although the load process is controlled in detail through use of this language (described later), most programmers do not require this degree of flexibility, and the manual page `ld`(1) in *A/UX Command Reference* will provide them with sufficient instruction in the use of this command. This chapter is a reference to enable you to determine what `ld` has done to your code.

The command language allows the loader to

- specify the machine's memory configuration
- combine object file sections in particular fashions
- cause the files to be bound to specific addresses or within specific portions of memory
- define or redefine global symbols at load time

Using ld

To use the loader, give the following command:

`ld` [*options*] *filename* ...

Files passed to `ld` must be object files, archive libraries containing object files, or text source files containing `ld` directives. `ld` uses the file's ***magic number*** (the first 2 bytes of the file) to determine which type of file it is encountering. If `ld` does not recognize the magic number, it assumes the file is a text file containing `ld` directives and attempts to parse it.

Input object files and archive libraries of object files are loaded together to form an output object file. If there are no unresolved references, the file should be executable. For additional information, see the section "Object Files" later in this chapter.

Object files have the form *name*`.o` throughout the examples in this chapter. The names of actual input object files need not follow this convention.

If you merely want to load the object files *file1*`.o` and *file2*`.o`, this command is enough:

`ld` *file1*`.o` *file2*`.o`

No directives to `ld` are needed. If no errors are encountered during the load, the output is left in the default file `a.out`.

The input file sections are combined in order. That is, if each of *file1*`.o` and *file2*`.o` contains the standard sections `.text`, `.data`, and `.bss`, the output object file also contains these three sections. The output `.text` section is a concatenation of `.text` from *file1*`.o` and *file2*`.o`. The `.data` and `.bss` sections are formed similarly. The output `.text` section is then bound at address 0x000000. The output `.data` and `.bss` sections are loaded together into contiguous addresses.

Instead of entering the names of files to be loaded, or entering `ld` options on the `ld` command line, you can place this information in a separate file and simply pass the file to `ld`. Such an input file containing loader directives is referred to as an *i-file* in this chapter. Its usefulness is explained in the paragraphs that follow. An i-file named `default.ld` is searched for automatically in the list of library directories (see the `-l` and `-L` options under "Options"). The default directory for this search is `/usr/lib`.

For example, if you frequently load the object files *file1*`.o`, *file2*`.o`, and *file3*`.o` with the same options *f1* and *f2*, you could enter the command

`ld` *-f1 -f2 file1*`.o` *file2*`.o` *file3*`.o`

each time you have to invoke `ld`. Alternatively, you could create an i-file containing the statements

-f
-f2
file1.o
file2.o file3.o

and use the following command:

`ld` *i-file*

Note that it is permissible to specify some of the object files to be loaded in the i-file and to specify others on the command line, as well as specifying some options in the i-file and others on the command line. Note also that either white space or newlines can separate the statements in an i-file. Input object files are loaded in the order they are encountered, whether on the command line or in an i-file. As an example, if a command line were

`ld` *file1.o i-file file2.o*

and the i-file contained

file3.o
file4.o

the order of loading would be

1. *file1.o*
2. *file3.o*
3. *file4.o*
4. *file2.o*

Note from this example that an i-file is read and processed immediately upon being encountered in the command line.

Loader concepts

There are several concepts and definitions with which you should become familiar before you proceed further.

Memory configuration

The virtual memory of an A/UX system is, for purposes of allocation, partitioned into configured memory and unconfigured memory. **Configured memory** indicates a range of memory for which the appropriate single in-line memory modules (SIMMs) have been installed and are available for use. **Unconfigured memory** denotes a range of memory for which no chips have been installed, or that is reserved by the operating system or the nuBus address space. The default is to treat all memory as configured.

- *Note:* Nothing can be loaded into unconfigured memory.

Specifying a certain memory range as unconfigured is one way of marking the addresses in that range as illegal or nonexistent with respect to the loading process. Memory configurations other than the default must be specified explicitly.

Unless otherwise specified, all discussion in this chapter of memory, addresses, and so on, concerns the configured sections of the address space.

Sections

A **section** of an object file is the smallest unit of relocation and must be a contiguous block of memory. You can identify a section with a starting address and a size. Information describing all the sections in a file is stored in **section headers** at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be **holes** or gaps between input sections (and between output sections), storage is allocated contiguously within each output section and may not overlap a hole in memory.

Addresses

The **physical address** of a section or symbol is the relative offset from address 0 of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is relative to address 0 of the virtual space, and the system performs another address translation.

Binding

You may need to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called **binding,** and the section in question is said to be "bound to" or "bound at" the required address. While binding is most commonly relevant to output sections, you can also bind global symbols with an assignment statement in the `ld` command language.

Object files

Object files are produced both by the assembler (typically as a result of calling the compiler) and by `ld`. `ld` accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to `ld` may also be absolute files (see the section "Nonrelocatable Input Files" for details).

Files produced by the compiler or assembler always contain three sections: (files using shared libraries contain additional sections):

`.text`	Contains the instruction text (for example, executable instructions).
`.data`	Contains initialized data variables.
`.bss`	Contains uninitialized data variables.

Files using shared libraries contain two additional sections:

`.init`	Contains shared library initialization fragments.
`.lib`	Contains the pathname to the shared library (for files using shared library executable files).

Files calling shared library executable files also contain dummy sections corresponding to the sections of the shared object file. For additional information, see Chapter 7, "Shared Libraries," in *A/UX Programming Languages and Tools,* Volume 1.

Here is an example of a typical (nonshared library) C program. If the source contained the following global declarations (not declared inside a function),

```
int i = 100;
char abc[200];
```

and the following assignment,

```
abc[i] = 0;
```

compiled code from the C assignment would be stored in `.text`, the variable `i` would be located in `.data`, and `abc` would be located in `.bss`.

There is an exception, however, to the rule: both initialized and uninitialized statics are allocated to the `.data` section (the value of an uninitialized static in a `.data` section is 0).

Options

You can intersperse options with filenames both on the command line and in an i-file. The ordering of options is not significant, except for the `l` and `L` options for specifying libraries.

The `l` option is shorthand notation for specifying an archive library, which is a collection of object files. Thus, as is the case with any object file, libraries are searched as they are encountered. The `L` specifies an alternative directory for searching for libraries. To be effective, an `-L` option must therefore appear before any `-l` options.

All options for `ld` must be preceded by a hyphen (-), whether in the i-file or on the `ld` command line. Options that have an argument (except for the `-l` and `-L` options) are separated from the argument by white space (blanks or tabs). Table 7-1 lists the supported options:

■ **Table 7-1** `ld` options

Option	Description
`-A`*factor*	Expands the default symbol table by the factor given.
`-F`	Performs the alignment necessary for demand paging. Sections will be aligned on stricter boundaries in the address space. Sections will be blocked in the output file so that they begin on file system block boundaries. In addition, the magic number 0413 will be stored in the file header.
`-L`*dir*	Changes the algorithm for searching for libraries to look in *dir* before looking in the default location. This option is used for `ld` libraries in the same way the `-I` option is for compiler `#include` files. The `-L` option is useful for finding libraries that are not in the standard library directory. To be useful, though, this option must appear before the `-l` option.
`-M`	Prints a warning message for all external variables that are multiply-defined.
`-N`	Adjusts the load point of the data section so that it will immediately follow the text section when loaded and stores the magic number 0407 in the header. This prevents the text from being shared (shared text is the default).
`-S`	Requests a silent `ld` run. All error messages from errors that do not immediately stop the `ld` run are suppressed.

(Continued)

■ **Table 7-1** `ld` options (Continued)

Option	Description
`-v`	Prints, on the standard error output, a *version id* identifying the version of `ld` involved.
`-vs`*num*	Takes *num* as a decimal version number identifying the `a.out` file that is produced. The version stamp is stored in the system header. This option is not directly recognized by the compiler (`cc`), so you must use the `-w` option to pass the version number to the loader; for example, `-Wl,-vs` *num* where `-w` is an option to `cc` allowing arguments to be passed, `l` stands for the loader (the arguments' destination), and `-vs`*num* are the arguments to `ld` that set the version number for the `a.out` file. Note that the space between `-vs` and *num* is required.
`-e`*ss*	Defines the primary entry point of the output file to be the symbol given by the argument *ss*.
`-f`*bb*	Sets the default fill value. The argument *bb* is a 2-byte constant. This value is used to fill holes formed within output sections. It is also used to initialize input `.bss` sections when they are combined with other non `.bss` input sections. If you don't use the `-f` option, the default fill value is 0 for all sections except the `.tv` section, whose default fill value is 0xFFFF.
`-ild`	Generates the sections reserved for use by the incremental loader. `-ild` invokes the `-r` option.
`-l`*file*	Specifies an archive library file as `ld` input. The argument *file* is a character string (less than ten characters) immediately following the `-l` without any intervening white space. As an example, `-lc` refers to `libc.a`, `-lC` to `libC.a`, and so on. The given archive library must contain valid object files as its members. The directory searched defaults to `usr/lib`, finding `usr/lib/libc.a`, `usr/lib/libC.a`, and so on. (See also the `-L` option.)
`-m`	Produces a map or list of the input/output sections (including holes) on the standard output.
`-o`*nn*	Names the output object file. The argument *nn* is the name of the A/UX system file to be used as the output file. The default output object filename is `a.out`. The option *nn* may be a full or partial A/UX pathname.
`-r`	Retains relocation entries in the output object file. Relocation entries must be saved if the output file is to be used as an input file in a subsequent `ld` call. If the `-r` option is used, unresolved references do not prevent the creation of an output object file (such a file is not executable, of course).

(Continued)

■ **Table 7-1** `ld` options (Continued)

Option	Description
`-s`	Strips line number entries and symbol table information from the output object file. Because relocation entries (`-r` option) are meaningless without the symbol table, you may not use `-r` if you use `-s`. All symbols are stripped, including global and undefined symbols.
`-t`	Disables checking of all instances of a multiply-defined symbol to be sure they are the same size.
`-u`*sym*	Introduces an unresolved external symbol into the output file's symbol table. The argument *sym* is the name of the symbol. This option is useful for loading entirely from a library, since the symbol table is initially empty and an unresolved reference is needed to force the loading of an initial routine from the library.
`-x`	Does not preserve any local (nonglobal) symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.
`-z`	Catches references through `NULL` pointers. The `z` is a mnemonic for "Do not place anything in address 0." This option is overridden if any section or memory directives are used.

The `ld` command language

The command language of `ld` allows you control over all phases of the loading process. Typically, `ld` operates on files created by `as` without needing your intervention. However, you can write your own program specifying how `ld` is to manipulate the components of object files.

Input to `ld` is a series of directives that together have the effect of combining various reloctable input object files, binding all objects to known addresses, and resolving object references so the resulting output object file is self-consistent and executable.

Expressions

Expressions can contain global symbols, constants, and most of the basic C language operators (see the last section of this chapter, "Syntax Diagram for Input Directives"). Constants in `ld` are defined as in C, with a number recognized as decimal unless preceded with `0` for octal or `0x` for hexadecimal.

◆ *Note:* All numbers are treated as type `long int`.

Symbol names may contain uppercase or lowercase letters, digits, and the underscore (_). Symbols within an expression have the value of the address of the symbol only. `ld` does not perform a symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, and so on.

`ld` uses a `lex`-generated input scanner to identify symbols, numbers, operators, and so forth. The current scanner design makes the following names reserved and unavailable as symbol or section names:

```
ALIGN       DSECT       MEMORY      PHY         SPARE
ASSIGN      GROUP       NOLOAD      RANGE       TV
BLOCK       LENGTH      ORIGIN      SECTIONS

align       group       length      origin      spare
assign      l           o           phy
block       len         org         range
```

Supported operators are are shown in order of precedence in Table 7-2:

■ **Table 7-2** Precedence of operators

Operator symbols					
!	+	- (unary minus)			
*	/	%			
+	- (binary minus)				
>>	<<				
==	!=	>	<	<=	>=
&					
|					
&&					
||					
=	+=	-=	*=	/=	

These operators have the same meaning as in the C language. Precedence decreases from the top to the bottom of the table. Operators on the same line have the same precedence.

Assignment statements

External symbols can be defined and assigned addresses via the *assignment statement*. The syntax of the assignment statement is

symbol = expression;

or

symbol op = expression;

where *op* is one of the operators +, -, *, or /.

◆ *Note:* Assignment statements must terminate with a semicolon.

All assignment statements (with one exception, described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input file–defined symbols are appropriately relocated, but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References to symbols given a value through an assignment statement within text and data access this latest assigned value. Assignment statements are processed in the same order in which they are input to `ld`.

Assignment statements are normally placed outside the scope of any section-definition directives (see the section "Section Definition Directives"). There is a special symbol, *dot* (`.`), however, that may occur only within a section-definition directive. This symbol refers to the current address of the `ld` location counter. Thus, assignment expressions involving dot are evaluated during the allocation phase of `ld`.

Assigning a value to the dot (`.`) symbol within a section-definition directive will increment or reset the `ld` location counter and may create holes within the section (as described in "Section Definition Directives").

Assigning the value of dot to a conventional symbol permits the final allocated address of a particular point within the load run to be saved.

`align` is provided as a shorthand notation to allow you to align a symbol to an *n*-byte boundary within an output section, where *n* is a power of 2. For example, the expression

`align(`*n*`)`

is equivalent to

`(. + `*n*` - 1) & (`*n*` - 1)`

Loader expressions can have either an absolute or a relocatable value, corresponding to a **type** of absolute or relocatable. When `ld` creates a symbol through an assignment statement, the symbol's value takes on the type of the expression. That type depends on the following rules:

- An expression with a single relocatable symbol (and 0 or more constants or absolute symbols) is relocatable. The value is in relation to the section of the referenced symbol.
- All other expressions have absolute values.

Specifying a memory configuration

MEMORY directives are used to specify

- the total size of the virtual space of an A/UX system
- the configured and unconfigured areas of the virtual space

If you do not supply any directives, `ld` assumes that all memory is configured. A/UX-capable Macintosh computers have a minimum of 4 MB of RAM, and use a virtual memory space of 4 GB. Generally, the only reason you would want to specify MEMORY directives for an A/UX application is to run it explicitly in physical, rather than virtual, memory space.

Using MEMORY directives, you can assign an arbitrary name of up to eight characters to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specified memory areas. Memory names may contain uppercase or lowercase letters, digits, and the three special characters `$`, `.`, or `_`. Names of memory ranges are used only by `ld` and are not carried in the output file symbol table or headers.

◆ *Note:* When you use MEMORY directives, all virtual memory that is not described in a MEMORY directive is considered to be unconfigured. `ld` does not use unconfigured memory in the allocation process, hence nothing can be loaded, bound, or assigned to an address within unconfigured memory.

As an option on the MEMORY directive, you may associate **attributes** with a named memory area. This restricts the memory areas (with specific attributes) to which an output section may be bound. The attributes you assign to output sections are recorded in the appropriate section headers in the output file to allow for possible error checking in the future. For example, putting a text section into writable memory is one potential error condition. Currently, error checking of this type is not implemented.

The attributes currently accepted are

R readable memory

`W` writable memory

`X` executable (instructions may reside in this memory)

`I` initializable (stack areas are typically not initialized)

Other attributes may be added in the future if necessary. If you do not specify any attributes on a `MEMORY` directive or if you do not supply any `MEMORY` directives, memory areas assume all of the attributes of `W`, `R`, `I`, and `X`.

The syntax of the `MEMORY` directive is

```
MEMORY
{
  name (attr) : origin = virt-addr[,] length = mem-length
  ...
}
```

The keyword `origin` (or `org` or `o`) must precede the origin of a memory range, and `length` (or `len` or `l`) must precede the length, as shown in the preceding prototype. The `origin` operand refers to the virtual address of the memory range. `origin` and `length` are entered as long integer constants in decimal, octal, or hexadecimal (standard C syntax). Origin and length specifications, as well as individual `MEMORY` directives, may be separated by white space or a comma.

By specifying `MEMORY` directives, you can tell `ld` that memory is configured in some manner other than the default. For example, if you need to prevent anything from being loaded to the first 0x10000 words of memory, you can do so with a `MEMORY` directive:

```
MEMORY
{
  valid : org = 0x10000, len = 0xFE0000
}
```

Region directives

This implementation of A/UX does not support region specifications, which are usually used only when developing UNIX® kernels.

Section definition directives

You can use the SECTIONS directive to describe how input sections are to be combined, to direct where output sections should be placed (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections. Sections in `ld` are equivalent to segments in `as`.

In the default case (where no SECTIONS directives are given), all input sections of the same name appear in an output section of that name. For example, if a number of object files from the compiler are loaded, each containing the three sections `.text`, `.data`, and `.bss`, the output object file will also contain three sections, `.text`, `.data`, and `.bss`. If two object files are loaded, one containing sections *s1* and *s2*, the other containing sections *s3* and *s4*, the output object file will contain the four sections *s1*, *s2*, *s3*, and *s4*. The order of these sections depends on the order in which the loader sees the input files.

The basic syntax of the SECTIONS directive is

```
SECTIONS
{
      secname :
      {
            file-specification ...,
            assignment-statement ...
      }
        ...
}
```

The various types of section definition directives are discussed in the remainder of this section.

File specifications

Within a section definition, the files and file sections to be included in the output section are listed in the order in which they are to appear. Sections from an input file are specified by

filename (*secname* ...)

Sections of an input file are separated by white space or commas (or both), as are the file specifications themselves.

If a filename appears with no sections listed, all sections from the file are loaded into the current output section; for example,

```
SECTIONS
{
     outsec1:
     {
          file1.o (sec1)
          file2.o
          file3.o (sec1, sec2)
     }
}
```

The order in which the input sections appear in the output section *outsec1* is given by

1. Section *sec1* from file *file1*.o.
2. All sections from *file2*.o, in the order they appear in the file.
3. Section *sec1* from file *file3*.o, then section *sec2* from file *file3*.o.

If any additional input files contain input sections named *outsec1*, these sections are loaded following the last section named in the *outsec1* definition. If there are any other input sections in *file1*.o or *file3*.o, they will be placed in output sections with the same names as the input sections.

Loading a section at a specified address

You may want to bond an output section to a specific virtual address to take advantage of a particular paging efficiency. This can be done as shown in the following SECTIONS directive example:

```
SECTIONS
{
     outsec addr:
     {
          file-spec (secname)
     }
     ...
}
```

addr is the bonding address, expressed as a C constant. If *outsec* does not fit at *addr* (perhaps because of holes in the memory configuration or because *outsec* is too large to fit without overlapping some other output section), ld issues an appropriate error message.

As long as output sections do not overlap and there is enough space, they may be bound anywhere in configured memory. The SECTIONS directives that define output sections do not have to be given to ld in any particular order.

`ld` does not ensure that each section's size consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4. Although it is not recommended, you can use the `ld` directives to force a section to start on an odd byte boundary, if unforeseen circumstances force you into this solution. If a section starts on an odd byte boundary, the section's contents either are accessed incorrectly or are not executed properly. If you specify an odd byte boundary, `ld` will issue a warning message.

Aligning an output section

You may request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The `ALIGN` option of the `SECTIONS` directive performs this function, so that the option

`ALIGN(`*n*`)`

is equivalent to specifying a bonding address of

`( . + `*n*` - 1) & (`*n*` - 1)`

- *Note:* This `ALIGN` option is different than the `align` option discussed in the section "Assignment Statements." `ALIGN` binds sections to an address boundary, while `align` binds a specific object to an address boundary.

You should note that the `as` assembler always pads the sections it generates to a full word length, making explicit alignment specifications unnecessary. This also holds true for the compilers `c89` and `cc`.

As an example of section alignment,

```
SECTIONS
{
        outsec  ALIGN(0x20000):
        {
            file-spec  (secname)
        }
        ...
}
```

The output section *outsec* is not bound to any given address, but is loaded to some virtual address that is a multiple of 0x20000 (for example, at address 0x0, 0x20000, 0x40000, 0x60000, and so on).

The default section alignment action for `ld` on MC68020 systems is to align the code (`.text`) at byte 0 and the data (`.data` and `.bss` combined) at the 4-megabyte boundary (byte 10487576). Since MMU requirements vary from system to system, alignment is not always desirable. The version of `ld` for MC68020 systems, therefore, provides a mechanism to allow the specification of different section alignments for each system, allowing you to align each section separately on
n-byte boundaries, where *n* is a multiple of 512.

The default allocation algorithm for `ld` is

1. Load all input `.text` sections together into one output section. This output section is called `.text` and is bound to an address of 0x0.
2. Load all input `.data` sections together into one output section. This output section is called `.data` and is bound to an address aligned to a machine-dependent constant.
3. Load all input `.bss` sections together into one output section. This output section is called `.bss` and is allocated so as to follow the output section `.data` immediately. Note that the output section `.bss` is not given any particular address alignment.

Specifying any `SECTIONS` directives results in this default allocation not being performed.

When all input files have been processed (and if no override is provided), `ld` will search the list of library directories (as with the `-l` flag option) for a file named `default.ld`. If this file is found, it is processed as an `ld` instruction file (or i-file). The `default.ld` file should specify the required alignment as outlined in the following paragraphs. If it does not exist, the default alignment action will be taken.

The `default.ld` file should appear as in the following example, with *align-value* replaced by the alignment requirement in bytes. The default allocation of `ld` is equivalent to supplying the following directive:

```
SECTIONS
{
    .text    : { }
    GROUP ALIGN(align-value) :
    {
         .data     : { }
         .bss      : { }
    }
}
```

where *align-value* is a machine-dependent constant.

◆ *Note:* The current (MC68020) system requires a *data rounding* of 2 MB. This requirement is subject to change as systems evolve.

The GROUP directive ensures that the two output sections, `.data` and `.bss`, are allocated (grouped) together. Bonding or alignment information is supplied only for the group, and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If you wish to place `.text`, `.data`, and `.bss` in the same segment, you should use the following SECTIONS directive:

```
SECTIONS
{
    GROUP           :
    {
        .text       : { }
        .data       : { }
        .bss        : { }
    }
}
```

Note that there are still three output sections (`.text`, `.data`, and `.bss`), but they are now allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive. To bind to 0xC0000, use

```
GROUP 0xC0000: {
```

To align to 0x10000, use

```
GROUP ALIGN(0x10000): {
```

With this addition, first the output section `.text` is bound at 0xC0000 (or is aligned to 0x10000); the remaining members of the group are allocated into the next available memory locations in order of their appearance.

When the GROUP directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
    .text: { }
    .data ALIGN(0x20000): { }
    .bss: { }
}
```

The `.text` section starts at virtual address 0x0 and the `.data` section at a virtual address aligned to 0x20000. The `.bss` section follows immediately after the `.text` section, but only if there is enough space. If there is not, it follows the `.data` section.

The order in which output sections are defined to `ld` cannot be used to force a certain allocation order in the output file.

Files that need to load in a shared library have the `.init` and `.text` sections grouped together. In the final stage of loading, the `.init` section becomes part of the `.text` section.

Creating holes within output sections

The special symbol dot (`.`) appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, `.` causes the `ld` location counter to be incremented or reset, and a hole is left in the output section.

Holes that are built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character, 0x00, or a supplied fill character). See the definition of the `-f` option in the section "Options" and the discussion of filling holes in the section "Initialized Section Holes or `.bss` Sections" later in this chapter.

Consider the following section definition:

```
SECTIONS
{
outsec:
     {
          . += 0x1000;
          f1.o (.text)
          . += 0x100;
          f2.o (.text)
          . = align (4);
          f3.o (.text)
     }
}
```

The effect of this command is as follows:

1. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input file *f1*`.o(.text)` is loaded after this hole.
2. The text of input file *f2*`.o` begins at 0x100 bytes following the end of *f1*`.o(.text)`.
3. The text of *f3*`.o` is loaded to start at the next full word boundary following the text of *f2*`.o` with respect to the beginning of *outsec*.

For the purposes of allocating and aligning addresses within an output section, `ld` treats the output section as if it began at address 0. If, in the above example, *outsec* is ultimately loaded to start at an odd address, the part of *outsec* built from *f3*`.o(.text)` also starts at an odd address, even though *f3*`.o(.text)` is aligned to a full word boundary. You can prevent this result by specifying an alignment factor for the entire output section:

```
outsec ALIGN(4): {
```

Expressions that decrement . are illegal. For example, subtracting a value from the location counter is not allowed, since overwrites are not allowed. The most common operators in expressions that assign a value to . are += and `align`.

Creating and defining symbols at loading time

You can use the assignment instruction of `ld` to give symbols a value that is loading dependent. Typically, there are three types of assignments:

- use of . to adjust the `ld` location counter during allocation
- use of . to assign an allocation-dependent value to a symbol
- assignment of an allocation-independent value to a symbol

The first case was discussed in the previous section. The second case provides a means to assign addresses (known only after allocation) to symbols; for example,

```
SECTIONS
{
     outsc1: {file-spec (secname) }
     outsc2:
     {
          file1.o (s1)
          s2_start = . ;
          file2.o (s2)
          s2_end = . - 1;
     }
}
```

The symbol *s2_start* is defined to be the address of *file2.o (s2)*, and *s2_end* is the address of the last byte of *file2.o (s2)*. Consider the following example:

```
SECTIONS
{
     outsc1:
     {
          file1.o (.data)
          mark = .;
          . += 4;
          file2.o (.data)
     }
}
```

In this example, the symbol `mark` is created and is equal to the address of the first byte beyond the end of the *file1.o* `.data` section. Four bytes are reserved for a future run-time initialization of the symbol `mark`. The type of the symbol is a long integer (32 bits).

Assignment instructions involving . must appear within SECTIONS definitions, since they are evaluated during allocation. Assignment instructions that do not involve . can appear within SECTIONS definitions, but typically do not. Such instructions are evaluated after allocation is complete.

It is risky to reassign a defined symbol to a different address. For example, if a symbol within .data is defined, initialized, and referenced within a set of object files being loaded, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. The associated initialized data are not moved to the new address. ld issues warning messages for each defined symbol that is being redefined within an i-file. Assignments of absolute values to new symbols are safe, however, because there are no references or initialized data associated with the symbol.

Allocating a section into named memory

The loader provides a mechanism for allowing you to specify a section to be loaded somewhere within a specific, named memory area (as previously specified on a MEMORY directive) using the > operator. The > notation is borrowed from the UNIX system concept of redirected output. For example,

```
MEMORY
{
     mem1:          o=0x000000    l=0x10000
     mem2 (RW):     o=0x020000    l=0x40000
     mem3 (RW):     o=0x070000    l=0x40000
     mem1:          o=0x120000    l=0x04000
}

SECTIONS
{
     outsec1: {f1.o(.data) } > mem1
     outsec2: {f2.o(.data) } > mem3
}
```

This code fragment directs ld to place *outsec1* at the first location within the memory area named *mem1* that is large enough to hold the section (somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FF). The *outsec2* is to be placed similarly in the address range 0x70000-0xAFFFF.

Initialized section holes or .bss sections

When holes are created within a section (as in the example in the section "Creating Holes Within Output Sections"), ld normally puts out bytes of zero as *fill*. By default, .bss sections are not initialized at all; that is, no initialized data, not even 0s, are generated for any .bss section by the assembler, nor are they supplied by the loader.

You can use initialization options in a SECTIONS directive to set such holes or to set .bss sections to an arbitrary 2-byte pattern.

◆ *Note:* Such initialization options apply only to .bss sections or holes.

For example, in an application you might want an uninitialized data table to be initialized to a constant value, without recompiling the .o file or filling a hole in the text area with a transfer to an error routine. You could specify that either specific areas within an output section or the entire output be initialized. Because no text is generated for an uninitialized .bss section, however, the entire section is initialized if part of such a section is initialized.

In other words, if a .bss section is to be combined with a .text or .data section (both of which are initialized), or if part of an output .bss section is to be initialized, one of the following will hold:

- Explicit initialization options must be used to initialize all .bss sections in the output section.
- ld will use the default fill value to initialize all .bss sections in the output section.

Consider the following ld i-file:

```
SECTIONS
{
    sec1:
    {
        f1.o (.text)
        . += 0x200;
        f2.o (.text)
    } = 0xDFFF
    sec2:
    {
        f1.o (.bss)
        f2.o (.bss)
    } = 0x1234
    sec3:
    {
        f3.o (.bss)
        ...
    } = 0xFFFF
    sec4: {f4.o (.bss)}
}
```

In the example above, the 0x200 byte hole in section *sec1* is filled with the value `0xDFFF`. In section *sec2*, *f1*`.o(.bss)` is initialized to the default fill value of `0x00`, and *f2*`.o(.bss)` is initialized to `0x1234`. All `.bss` sections within *sec3* as well as all holes are initialized to `0xFFFF`. Section *sec4* is not initialized; that is, no data are written to the object file for this section.

Notes and special considerations

The following sections are collections of additional information you may find helpful in understanding the loader.

Using archive libraries

Each member of an archive library (for example, `libc.a`) is a complete object file, typically consisting of the standard three sections:

- `.text`
- `.data`
- `.bss`

Shared library archives contain one or two (optional) additional sections:

- `.init`
- `.lib`

In addition to these sections, files calling on shared library executable files contain dummy sections corresponding to sections of the shared object. For further information, see Chapter 7, "Shared Libraries," in *A/UX Programming Languages and Tools*, Volume 1.

Archive libraries are created through the use of the A/UX system `ar` command on object files generated by running `cc` or `as`. Shared libraries are created using the `mkshlib` command. An archive library is always processed using *selective inclusion:* only those members that resolve existing undefined-symbol references are taken from the library for loading.

Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for loading whenever the following conditions exist:

- A reference to a symbol is defined in that member.
- The reference is found by `ld` prior to the actual scanning of the library.

When a library member is included by searching the library inside a SECTIONS directive, all input sections from the member are included in the output section being defined.

When a library member is included by searching the library outside a SECTIONS directive, all input sections from the member are included in the output section with the same name. That is, the `.text` section of the member goes into the output section named `.text`, the `.data` section of the member into `.data`, the `.bss` section of the member into `.bss`, and so on. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that

- Specific members of a library may not be referenced explicitly in an i-file.
- The default rules for the placement of members and sections may not be overridden when they apply to archive library members.

The `-l` option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. They do not, however, have to be. Furthermore, you can specify archive libraries without using the `-l` option simply by giving the full or relative A/UX system pathname.

◆ *Note:* The ordering of archive libraries is important, because a member extracted from the library must satisfy a reference that is known to be unresolved at the time the library is searched.

You can specify archive libraries more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. `ld` will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Because it runs on the Macintosh II, `ld` uses a random-access library. All machines running a pre–System V UNIX system use an old format library that must be searched linearly.

The loader will make one search through a library in the old format, but will continue to search through a library in the new format until it has determined that it can resolve no more references from that library. Because of the different searching algorithms used, programs that are ported from pre–System V UNIX machines can include files from libraries in a different order.

Be careful when using archive libraries in a subsystem loading environment. If a member of an archive (an object file) is to be included in a subsystem final load file, there must be a reference within the subsystem being loaded to a symbol defined in that object file. You can use the `-u` option to create unresolved references that force the loading of archive members. Consider the following example:

- The input files *file1*`.o` and *file2*`.o` each contain a reference to the external function FCN.

- Input *file1*.o contains a reference to symbol ABC.
- Input *file2*.o contains a reference to symbol XYZ.
- Library liba.a, member 0, contains a definition of XYZ.
- Library libc.a, member 0, contains a definition of ABC.
- Both libraries have a member 1 that defines FCN.

Depending on the order in which files and libraries appear on the command line, different library members can be included for loading. If the ld command is entered as

```
ld file1.o -la file2.o -lc
```

the FCN references are satisfied by liba.a, member 1; ABC is obtained from libc.a, member 0; and XYZ remains undefined (because the library liba.a is searched before *file2*.o is specified). If the ld command is entered as

```
ld file1.o file2.o -la -lc
```

the FCN references are satisfied by liba.a, member 1; ABC is obtained from libc.a, member 0; and XYZ is obtained from liba.a, member 0. If the ld command is entered as

```
ld file1.o file2.o -lc -la
```

the FCN references are satisfied by libc.a, member 1; ABC is obtained from libc.a, member 0; and XYZ is obtained from liba.a, member 0.

You can use the -u option to force the loading of library members when the loading run does not contain an actual external reference to the members. For example,

```
ld -u routl -la
```

creates an undefined symbol called routl in the ld global symbol table. If any member of library liba.a defines this symbol, it, and perhaps other members as well, is extracted. Without the -u option, there would have been no trigger to cause ld to search the archive library.

Dealing with holes in physical memory

When memory configurations are defined such that unconfigured areas exist in virtual memory, each application or user has the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
      mem1:    o = 0x00000    l = 0x02000
      mem2:    o = 0x40000    l = 0x05000
      mem3:    o = 0x20000    l = 0x10000
}
```

Let the files *f1*.o, *f2*.o, ... *fn*.o each contain the standard three sections `.text`, `.data`, and `.bss`, and let the combined `.text` section be 0x12000 bytes. There is no configured area of memory into which this section may be placed. Appropriate directives must be supplied to break up the `.text` output section so `ld` can do allocation. For example,

```
SECTIONS
{
    txt1:
    {
        f1.o (.text)
        f2.o (.text)
        f3.o (.text)
    }
    txt2:
    {
        f4.o (.text)
        f5.o (.text)
        f6.o (.text)
    }
    ...
}
```

Allocation algorithm

An output section is formed either as a result of a `SECTIONS` directive or by combining input sections of the same name. An output section can be made up of 0 or more input sections. After an output section's composition is determined, it must be allocated into configured virtual memory. `ld` uses an algorithm that attempts to minimize fragmentation of memory, which increases the possibility that a loading run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Allocate any output sections for which explicit bonding addresses are specified.
2. Allocate any output sections to be included in a specified memory area. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory area, taking into consideration any alignment.
3. Allocate output sections that are not handled by steps 1 or 2.

If all memory is contiguous and configured (the default), and no SECTIONS directives are given, output sections are allocated in the order they appear to ld, normally .text, .data, .bss. Otherwise, output sections are allocated, in the order they were defined or made known to ld, into the first available space they fit.

Incremental loading

As previously mentioned, the output of ld can be used as an input file to subsequent ld runs, provided that the relocation information is retained (using the -r option). With large applications you may find it desirable to partition C programs into subsystems, load each subsystem independently, and then load the entire application. For example,

Step 1:

```
ld -r  -o outfile1  i-file1

/* i-file1  */
SECTIONS
{
    ss1:
    {
        f1.o
        f2.o
        ...
        fn.o
    }
}
```

Step 2:

```
ld -r  -o outfile2  i-file2

/* i-file2  */
SECTIONS
{
    ss2:
    {
        g1.o
        g2.o
        ...
        gn.o
    }
}
```

Step 3:

```
ld  -a -m -o final.out outfile1 outfile2
```

By judiciously forming subsystems, applications can achieve a form of *incremental loading,* whereby it is necessary to reload only a portion of the total load when a few programs are recompiled.

To apply this technique, follow two simple rules:

1. Intermediate loads must contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. You should not do any binding of output sections in these runs.
2. All allocation and memory directives, as well as any assignment statements, must be included in the final `ld` call only.

DSECT, COPY, and NOLOAD sections

You can give sections a type in a section definition, as shown in the following example:

The DSECT option creates a *dummy section,* which has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map (the `-m` option) generated by `ld`.
2. It can overlay other output sections and even unconfigured memory. dummy sections may overlay other dummy sections.
3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the dummy section were actually loaded at its virtual address. Other input sections may reference DSECT-defined symbols. Undefined external symbols found within a dummy section cause specified archive libraries to be searched; any members that define such symbols are loaded normally (not in the dummy section or as a dummy section).
4. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

A **copy section** is created by the COPY option. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information are written to the output file.

A **noload section** is allocated virtual space, appears in the memory map, and so forth. A section of the type NOLOAD differs in only one respect from a normal output section: text and data are not written to the output file.

As an example:

```
SECTIONS
{
    name1 0x200000 (DSECT)    : {file1.o}
    name2 0x400000 (COPY)     : {file2.o}
    name3 0x600000 (NOLOAD)   : {file3.o}
}
```

Here, none of the sections from *file1*.o are allocated, but all symbols are relocated as though the sections were loaded at the specified address. Other sections could refer to any of the global symbols and are resolved correctly.

Output file blocking

You can use two options to affect the physical file offsets of the information written to the output file by `ld`:

- The `BLOCK` option permits any output section to be aligned in the output field at a specified
  n-byte boundary.
- The `-B` option causes padding sections to be generated in the output file.

Both features are provided explicitly for the use of `ldp`, which constructs *pfiles* for DMERT. The output sections of a *pfile* have certain requirements in terms of physical file offsets. These requirements can be met using `BLOCK` and `-B`.

You can apply the `BLOCK` option to any output section or `GROUP` directive. It directs `ld` to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the loading process. It is used purely to adjust the physical position of the section in the output file.

```
SECTIONS
{
    .text BLOCK(0x200):{ }
    .data ALIGN(0x20000)BLOCK(0x200):{ }
}
```

In this `SECTIONS` directive example, `ld` assures that each section, `.text` and `.data`, is physically written at a file offset that is a multiple of 0x200 (for example, at an offset of 0, 0x200, 0x400, ..., and so on, in the file).

Nonrelocatable input files

If you intend to use a file produced by `ld` in a subsequent `ld` run, you should set the `-r` option for the first `ld` run. This preserves relocation information and permits the sections of the file to be relocated by the subsequent `ld` run.

When `ld` detects an input file that does not have relocation or symbol table information, it gives a warning message. Such information may be removed by `ld` (see the `-s` option in the section "Options") or by the `strip`(1) program. Note, however, that the loading run continues, using the nonrelocatable input file. For such a load to be successful (that is, actually and correctly to load all input files, relocate all symbols, resolve unresolved references, and so on), two conditions for the nonrelocatable input files must be met:

1. Each input file must have no unresolved external references.
2. Each input file must be bound to the same virtual address as it was in the `ld` run that created it.

Note that if these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this restriction, you must take extreme care when supplying such input files to `ld`.

The `-ild` option

When the `-ild` option is used, the loader creates a pair of dummy sections of type `DSECT` for each unallocated, configured area of memory. These dummy sections have unique names in the form of `.i_l_d`*nn*, where *nn* is a 2-digit decimal integer in the range from 00 through 99. At most, 50 pairs of these sections will be created by the loader. These sections identify the boundaries of the unused memory space and are similar to `.bss` sections in that they do not contain any text or initialized data. The loader also creates a dummy section named `.history`. These sections are used later by the incremental loader.

Error messages

The following sections report the error messages you may receive from `ld`. The sections are arranged by general topic.

Corrupt input files

Certain error messages indicate that the input file is corrupt, nonexistent, or unreadable. If you get any of them, you should check that the file is in the correct directory with the correct permissions. If the object file is corrupt, try recompiling or reassembling it. These error messages include

`Can't open` *name*

`Can't read archive header from archive` *name*

`Can't read file header of archive` *name*

`Can't read 1st word of file` *name*

`Can't seek to the beginning of file` *name*

`Fail to read file header of` *name*

`Fail to read lnno of section` *sect* `of file` *name*

`Fail to read magic number of file` *name*

`Fail to read section headers of file` *name*

`Fail to read section headers of library` *name* `member` *number*

`Fail to read symbol table of file` *name*

`Fail to read symbol table when searching libraries`

`Fail to read the aux entry of file` *name*

`Fail to read the field to be relocated`

`Fail to seek to symbol table of file` *name*

`Fail to seek to symbol table when searching libraries`

`Fail to seek to the end of library` *name* `member` *number*

`Fail to skip aux entries when searching libraries`

`Fail to skip the mem of struct of` *name*

`Illegal relocation type`

`No reloc entry found for symbol`

`Reloc entries out of order in section` *sect* `of file` *name*

`Seek to` *name* `section` *sect* `failed`

Seek to *name* section *sect* lnno failed

Seek to *name* section *sect* reloc entries failed

Seek to relocation entries for section *sect* in file *name* failed.

Errors during output

Certain errors occur because `ld` cannot write to the output file. This usually indicates that the file system is out of space. Messages to this effect include

Cannot complete output file *name*. Write error.

Fail to copy the rest of section *num* of file *name*

Fail to copy the bytes that need no reloc of section *num* of file

name I/O error on output file *name*.

Internal errors

Certain messages indicate that something is wrong with `ld` internally. If you get them, there is probably nothing you can do except to get help from another experienced user of `ld`. Such messages include

Attempt to free nonallocated memory

Attempt to reinitialize the SDP aux space

Attempt to reinitialize the SDP slot space

Default allocation did not put .data and .bss into the same region

Failed to close SDP symbol space

Failure dumping an AIDFN*xxx* data structure

Failure in closing SDP aux space

Failure to initialize the SDP aux space

Failure to initialize the SDP slot space

Internal error: audit_groups, address mismatch

Internal error: audit_group, finds a node failure

Internal error: fail to seek to the member of *name*

Internal error: in allocate lists, list confusion (*num num*)

Internal error: invalid aux table id

Internal error: invalid symbol table id

Internal error: negative aux table ld

Internal error: negative symbol table id

Internal error: no symtab entry for DOT

Internal error: split_scns, size of *sect* exceeds its new displacement.

Allocation errors

Certain error messages appear during the allocation phase of the load. They generally appear if a section or group does not fit at a certain address or if the given MEMORY or SECTION directives conflict in some way. If you are using an i-file and get such messages, check that MEMORY and SECTION directives allow enough room for the sections to ensure that nothing overlaps and that nothing is being placed in unconfigured memory. For more information, see the sections "The ld Command Language" and "Notes and Special Considerations." These messages include

Bond address *address* for *sect* is not in configured memory

Bond address *address* for *sect* overlays previously allocated section *sect* at *address*

Can't allocate output section *sect*, of size *num*

Can't allocate section *sect* into owner *mem*

Default allocation failed: *name* is too large

GROUP containing section *sect* is too big

Memory types *name1* and *name2* overlap

Output section *sect* not allocated into a region

sect at *address* overlays previously allocated section *sect* at *address*

sect, bonded at *address*, won't fit into configured memory

sect enters unconfigured memory at *address*

Section *sect* in file *name* is too big.

Misuse of loader directives

Certain error messages are explanations that occur following the misuse of an input directive. If you get them, please review the appropriate section in this chapter. These messages, and brief explanations of their causes, follow.

Adding *name*(*sect*) to multiple output sections.
: The input section is mentioned twice in the SECTIONS directive.

Bad attribute value in MEMORY directive: *c*.
: The attribute *c* is illegal. An attribute must be one of R, W, X, or I.

Bad flag value in SECTIONS directive, *option*.
: Only the -l option is allowed inside of a SECTIONS directive.

Bad fill value.
: The fill value must be a 2-byte constant.

Bonding excludes alignment.
: The section will be bound at the given address, regardless of the alignment of that address.

Cannot align a section within a group
Cannot bond a section within a group
Cannot specify an owner for sections within a group
: The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group may not be handled individually.

DSECT *sect* can't be given an owner
DSECT *sect* can't be linked to an attribute.
: Because dummy sections do not participate in the memory allocation, it is meaningless for a dummy section to be given an owner or an attribute.

Region commands not allowed
: The A/UX loader does not accept the REGION commands.

`Section` *sect* `not built.`
The most likely cause of this is a syntax error in the `SECTIONS` directive.

`Semicolon required after expression`
`Statement ignored.`
This is caused by a syntax error in an expression.

`Usage of unimplemented syntax.`
The A/UX `ld` does not accept all possible commands.

Misuse of expressions

Certain errors arise from the misuse of an input expression. If you receive any of the following messages, please review the appropriate section in this chapter.

`Absolute symbol` *name* `being redefined.`
An absolute symbol may not be redefined.

`ALIGN illegal in this context.`
Alignment of a symbol may be done only within a `SECTIONS` directive.

`Attempt to decrement DOT`
`Illegal assignment of physical address to DOT.`
`Illegal operator in expression`
`Misuse of DOT symbol in assignment instruction.`
You may not use the dot symbol (`.`) in assignment statements that are outside of `SECTIONS` directives.

`Symbol` *name* `is undefined.`
All symbols referenced in an assignment statement must be defined.

`Symbol` *name* `from file` *name* `being redefined.`
A defined symbol may not be redefined in an assignment statement.

`Undefined symbol in expression.`
All symbols used in expressions must be defined.

Misuse of options

Certain errors arise from the misuse of options. If you get any of the following messages, please review the appropriate section of this book:

`Both -r and -s flags are set.`
`-s flag turned off.`
Further relocation requires a symbol table.

`Can't find library lib`*x*`.a`

`-L path too long (`*string*`)`

`-o file name too large (>128 char), truncated to (`*string*`)`

`Too many -L options, seven allowed.`

Some options require white space before the argument, some do not; see the section "Options." Including extra white space or not including the required white space is the most likely cause of the following messages:

option `flag does not specify a number`

option `is an invalid flag`

`-e flag does not specify a legal symbol name:` *name*

`-f flag does not specify a two-byte number:` *num*

`No directory given with -L`

`-o flag does not specify a valid file name:` *string*

`-l flag (specifying a default library) is not supported`

`-u flag does not specify a legal symbol name:` *name*.

Space constraints

Certain error messages can occur if `ld` attempts to allocate more space than is available. If you get them, you should attempt to decrease the amount of space used by `ld`. You can do this by making the i-file less complicated or by using the `-r` option to create intermediate files. These space constraint messages include

`Fail to allocate` *num* `bytes for slotvec table`

`Internal error: aux table overflow`

`Internal error: symbol table overflow`

`Memory allocation failure on` *num*`-byte call`

`Memory allocation failure on realloc call`

`Run is too large and complex.`

Miscellaneous errors

Errors occur for many reasons. If one occurs that has not been explained in a previous section, refer to the error message for an indication of where to look in this reference. Miscellaneous error messages include

`Archive symbol table is empty in archive` *`name`*`, execute 'ar ts` *`name`*`' to restore archive symbol table.`

On systems with a random-access archive capability, such as A/UX, the loader requires that all archives have a symbol table. This symbol table may have been removed by `strip`.

`Can't create intermediate ld file` *`name`*
`Can't open internal file` *`name`*
: These two messages are possible only when the loader uses two processes. This would indicate that the temp directory (usually `/tmp` or `/usr/tmp`) is out of space, or that the loader does not have permission to write in it.

`Cannot create output file` *`name`*.
: You may not have write permission in the directory where the output file is to be written.

`File` *`name`* `is of unknown type, magic number =` *`num`*
`Ifile nesting limit exceeded with file` *`name`*.
: i-files may be nested 16 deep.

`Library` *`name`*`, member has no relocation information.`
`Multiply defined symbol` *`sym`*`, in` *`name`* `has more than one size`
: A multiply-defined symbol has not been defined in the same manner in all files.

`name(sect)` `not found`
: An input section specified in a `SECTIONS` directive was not found in the input file.

`Section` *`sect`* `starts on an odd byte boundary!`
: This warning occurs only if you specifically bind a section at an odd boundary.

`Sections .text, .data or .bss not found;`

`Optional header may be useless.`
The system `a.out` header uses values found in the `.text`, `.data`, and `.bss` section headers.

`Line nbr entry (`*num num*`) found for nonrelocatable symbol:`
`Section` *sect*`, file` *name*
This error is generally caused by an interaction of `yacc`(1) and `cc`(1). See the section "Notes and Special Considerations."

`Undefined symbol` *sym* `first referenced in file` *name*.
Unless you use the `-r` option, `ld` requires that all referenced symbols be defined.

`Unexpected EOF (End Of File).`
There is a syntax error in the i-file.

Syntax diagram for input directives

Input to `ld` is a series of directives that together have the effect of combining various reloctable input object files, binding all objects to known addresses, and resolving object references so the resulting output object file is self-consistent and executable. Table 7-3 contains syntax diagrams for the input directives.

In Table 7-3, a particular notation is used. The terms on the left define the terms on the right. For example, the expansion

term → *directive1*
→ *directive2*

means that *term* can be made up of *directive1* or *directive2*.

- *Note:* Number suffixes have been added to some metalanguage terms to illustrate treatment of multiple arguments. You should ignore these suffixes when seeking the definition of such terms.

Ellipses (...) indicate that several of the elements on the right can comprise a left-hand element. For example, the expansion

term → *directive* ...

means that *term* is made up of one or more *directives*. Brackets indicate optional directives, and braces indicate that the contents must be included in the directive.

For *flags*, one or more blanks, tabs, or newlines can be substituted wherever there is a space between a flag option and its argument.

■ **Table 7-3** Directive expansion

Directive	→	Expanded directive
file	→	*cmd*...
cmd	→	*memory*
	→	*sections*
	→	*assignment*
	→	*filename*
	→	*flags*
memory	→	`MEMORY { ` *memory-spec1* `[ [,] ` *memory-spec2*`] ... }`
memory-spec	→	*name* `[` *attributes* `] :` *origin-spec* `[,]` *length-spec*
attributes	→	`( [ R ][ W ][ X ][ I ] )`
origin-spec	→	*origin* = *long*
length-spec	→	*length* = *long*
origin	→	`ORIGIN`
	→	`o[rigin]`
	→	`o[rg]`
length	→	`LENGTH`
	→	`l[ength]`
	→	`l[en]`
sections	→	`SECTIONS {` *sec-or-group* `...}`
sec-or-group	→	*section*
	→	*group*
	→	*library*

(Continued)

■ **Table 7-3** Directive expansion (Continued)

Directive	→	Expanded directive
section	→	*name sec-options* : { *statement-list* } [*fill*] [*mem-spec*]
sec-options	→	[*addr*] [*align-option*] [*block-option*] [*type-option*]
addr	→	*long*
align-option	→	*align* (*long*)
align	→	`ALIGN`
	→	`align`
block-option	→	*block* (*long*)
block	→	`BLOCK`
	→	`block`
type-option	→	`(DSECT)`
	→	`(NOLOAD)`
	→	`(COPY)`
statement-list	→	*statement1* [*statement2*] ...
statement	→	*filename* [(*name-list*)] [*fill*] *library assignment*
name-list	→	*name1* [[,] *name2*] ...
fill	→	= *long*
library	→	`-l` *name*
assignment	→	*lside assign-op expr end*
lside	→	*name*
	→	.

(Continued)

■ **Table 7-3** Directive expansion (Continued)

Directive	→	Expanded directive
assign-op	→	=
	→	+=
	→	-=
	→	*=
	→	/=
expr	→	*term*
	→	*expr binary-op expr*
term	→	*long*
	→	*name*
	→	*align* (*term*) [1]
	→	(*expr*)
	→	*unary-op term*
unary-op	→	!
	→	-
binary-op	→	*
	→	/
	→	%
	→	+
	→	-
	→	>>
	→	<<
	→	==
	→	!=
	→	>
	→	<
	→	<=
	→	>=
	→	&
	→	|
	→	&&
	→	||

(Continued)

[1]These appear to be circular references, but in practice they are (eventually) resolved by definition to a defined element.

■ **Table 7-3** Directive expansion (Continued)

Directive	→	Expanded directive
end	→	;
	→	,
group	→	GROUP *group_options* : { *section-list* } [*mem-spec*]
group-options	→	[*addr*] [*align-option*]
section-list	→	*section1* [[,] *section2*] ...
mem-spec	→	> *name*
	→	> *attributes*
flags	→	`-e` *name*
	→	`-f` *long*
	→	`-ild`
	→	`-l`*name*
	→	`-m`
	→	`-o` *filename*
	→	`-r`
	→	`-s`
	→	`-t`
	→	`-u` *name*
	→	`-x`
	→	`-z`
	→	`-F`
	→	`-L`*pathname*
	→	`-M`
	→	`-N`
	→	`-S`
	→	`-V`
	→	`-VS` *long*
name	→	Any valid symbol name.
long	→	Any valid long integer constant.
filename	→	Any valid A/UX operating system filename. This may include a full or partial pathname.
pathname	→	Any valid A/UX operating system pathname (full or partial).

Glossary

***:** C symbol denoting a pointer type.

absolute value: (1) An expression or identifier whose memory location (and value) is known at assembly time. Compare **relocatable value**. (2) The positive value of any number.

active window: The frontmost window on the desktop.

address: A number that specifies a location of an **object** in memory.

aggregate: A structure or array.

alert: A warning or report of an error, in the form of an alert box, sound from the Macintosh's speaker, or both.

alert box: A box that appears on the screen to give a warning or report an error during a Macintosh application.

alias: A different name for the same entity.

A-line instructions: Unimplemented 68000-family instructions, used by the Macintosh to implement Toolbox and Macintosh Operating System calls.

allocate: To reserve an area of memory for use.

anachronism: A language feature, dating back to the early days of C, that is no longer supported by most C compilers. Anachronisms are **obsolete.**

ANSI C: The C language as described in the *American National Standard for Information Systems—Programming Language C*, document number X3.159-1989.

ANSI C standard: The finalized standard on the C language as defined by the American National Standards Institute.

application: A program that can be launched from the Macintosh Finder. An application runs stand-alone and has the file type `'APPL'`.

application heap: The portion of the heap available to the running application program and the Toolbox.

application space: Memory that's available for dynamic allocation by applications.

array: A data structure containing an ordered set of elements.

ASCII: Acronym for *American Standard Code for Information Interchange,* a system of assigning code numbers to letters, numerals, punctuation marks, and control codes.

automatic variable: A dynamic local variable that comes into existence when a function is called or a compound statement scope is entered and that disappears when it is exited.

big-endian processor: A processor that puts the most significant byte of a word first. So named after a Lilliputian political dispute.

block: A group regarded as a unit; usually refers to data or memory in which data is stored.

block device: A device that reads and writes blocks of bytes at a time. It can read or write any accessible block on demand.

buffer: A holding area in RAM where information can be stored temporarily.

button: A standard Macintosh control that causes some immediate or continuous action when clicked or pressed with the mouse. See also **radio button.**

C: (1) A programming language originally designed by Dennis Ritchie in 1972 and expanded by the ANSI C standard.

C string: A sequence of characters terminated by a null byte.

calling conventions: The conventions under which a function is called, parameters are passed to it, and a result, if any, is returned. A/UX C supports both C-style and Pascal-style calling conventions.

carriage return (`\r`): A control code generated by the RETURN key.

`char`: An 8-bit character data type whose range is –128 to 127.

character device: A device that reads or writes a stream of characters, one at a time. It can neither skip characters nor go back to a previous character.

class specifier: A keyword, like `register`, that specifies a type's storage class.

cast: a construct causing an expression value to convert to a named data type.

closed file: A file without an access path. Closed files cannot be read from or written to.

code resource: A resource that contains a program's code—most commonly a resource of type `'CODE'` (for applications), but other resource types such as `'DRVR'` and `'PDEF'` also contain code.

code segment: An individual `'CODE'` resource, comprising part of the code of a Macintosh application. Segments are loaded in and out of memory by the segment loader.

command: In the Standard C Library, a parameter that tells a function which of several actions to perform; in an A/UX shell, a command name and parameters.

command file: A file consisting of executable commands that can be run from the shell. Also called a *script*.

compiler option: A symbol placed in the A/UX C command line to send an instruction to the compiler.

control character: A nonprinting character that controls or modifies the way information is printed or displayed.

`__DATE__`: A reserved preprocessor symbol that represents the current date at compile time.

data buffer: Heap space containing information to be written to a file or device driver from an application, or read from a file or device driver to an application.

data fork: The part of a file that contains data accessed via the File Manager.

deprecated: Not recommended. A feature is deprecated if it is **obsolescent**—that is, it may work on the current compiler, but will not work on future compilers.

dereference: To refer to a block by its master pointer instead of its handle.

desktop: The screen as a surface for doing work on the Macintosh.

device: A part of the Macintosh, or a piece of external equipment, that can transfer information into or out of the Macintosh.

device driver: A program that controls the exchange of information between an application and a device.

diagnostic output: The file to which A/UX tools, including the C compiler, write error messages and progress information. Diagnostic output appears following the commands being executed in the active window by default, and can be redirected to other files, windows, and selections. In C, diagnostic output is referenced using stream `stderr`.

dialog: Same as **dialog box.**

dialog box: A box that a Macintosh application displays to request information it needs to complete a command, or to report that it's waiting for a process to complete.

direct function: A function that makes a direct trap call to the Macintosh ROM.

directory: A subdivision of a volume that can contain files as well as other directories; equivalent to a folder.

`double`: A 64-bit floating-point data type—the IEEE `double` type.

ellipsis character: (1) The ANSI C ellipsis character (. . .) is three periods in the parameter list of a function prototype. (2) The Macintosh ellipsis character (...) is produced by typing OPTION-semicolon. The two are not interchangeable.

end-of-file: See **logical end-of-file** or **physical end-of-file.**

`enum`: An enumerated data type of 8, 16, or 32 bits, depending on the range of the enumerated literals.

environment: Consists of exported variables and signal-handling capabilities.

exception: An error or abnormal condition detected by the processor in the course of program execution; includes interrupts, traps, and floating-point exceptions.

exit function: A function that is registered with `onexit` for execution when the program terminates.

expression: An expression is a sequence of operators and operands the specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination of these.

external reference: A reference to a routine or variable defined in a separate compilation or assembly.

`__FILE__`: A reserved preprocessor symbol that represents the current filename at compile time.

file: A named, ordered sequence of bytes; a principal means by which data is stored and transmitted on the Macintosh.

file buffered: A buffering style in which characters sent to an output I/O function are queued and written as a block.

file control block: A fixed-length data structure, contained in the file-control-block buffer, where information about an access path is stored.

file descriptor: A file reference number returned by a `creat` or `open` call.

file directory: The part of a volume that contains descriptions and locations of all the files and directories on the volume. There are two types of file directories: hierarchical file directories and flat file directories.

File Manager: The part of the Macintosh Operating System that supports file I/O.

file pointer: A pointer to the next byte to be read or written in a **stream.**

`FILE` variable: A variable containing information about a stream, including the file descriptor and buffer size, location, and style.

filename: A sequence of up to 31 printing characters (excluding colons) that identifies a file. See also **pathname.**

Finder: The application that maintains the Macintosh desktop and launches other programs. The Finder is also the default startup application.

fixed-point number: A signed 32-bit quantity containing an integer part in the high-order word and a fractional part in the low-order word.

`float`: A 32-bit floating-point data type—the IEEE `single` type.

Floating-Point Arithmetic Package: A Macintosh package that supports extended-precision arithmetic according to IEEE Standard 754.

floating-point coprocessor (MC68881): A coprocessor on the Macintosh II that provides high-speed support for extended-precision arithmetic.

flush: To write out the contents of a buffer.

font: A complete set of characters of one typeface. A font may be restricted to a particular size and style, or may comprise multiple sizes, or multiple sizes and styles.

fork: One of the two parts of a file; see **data fork** and **resource fork.**

free block: A memory block containing space available for allocation.

function: A C routine, which may or may not return a value. Equivalent to a Pascal function or a Pascal procedure, respectively.

function prototype: A function declaration containing an argument list as specified by the ANSI standard. Function prototypes, one of the major features of ANSI C, allow stronger type checking by the compiler.

full pathname: A pathname beginning from the root directory. A full pathname is a pathname that contains embedded colons but no leading colon. Compare **partial pathname.**

global variable: A variable that is valid for all applications.

header file: A file whose contents will be included with the source file at compile time; it contains function declarations, macros, types, and `#define` directives used by the compiler. Also called an *include file.*

HFS: See **hierarchical file system.**

hierarchical file system (HFS): A system in which directories are used to hold files, applications, and other directories. HFS is used on hard disks and on 800K (and larger) floppy disks.

highlight: To display an object on the screen in a visually distinctive way, such as inverting it.

hybrid application: An application taking advantage of both the UNIX® and Macintosh features of A/UX. There are generally two kinds: (1) UNIX applications that make Macintosh toolbox calls, and (2) Macintosh applications that make A/UX system calls.

icon: A 32-by-32-bit image that graphically represents an object, concept, or message.

include : A XXX.

identifier: The name of an **object**, limited to 1024 characters.

index: A numeric value that indicates the position of an element in a sublist or array, expressed by a subscript.

`int`: A 32-bit integer data type whose range is –2,147,483,648 to 2,147,483,647. Identical to type `long` in A/UX.

integral: One of the types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`.

Integrated Environment: A library of functions that provide low-level I/O, access to font and tab settings associated with text files, and signal-handling capabilities. In C, the routines are part of the Standard C Library.

interface: The compile-time and run-time linkage between your C program and Pascal routines such as those documented in *Inside Macintosh.*

I/O: Abbreviation for *input* and *output* operations, taken collectively.

K&R C: The original version of the C language, as described in the first edition of Kernighan and Ritchie's *The C Programming Language.*

label: A

`__LINE__`: A reserved preprocessor symbol that represents the current line number in the current source file at compile time.

line buffered: A buffering style in which each line of output is saved for writing as soon as a newline character is written.

little-endian processor: A processor that puts the least significant byte of a word first. So named after a Lilliputian political dispute.

location counter: A

locked file: A file whose data cannot be changed.

logical block: Volume space composed of 512 consecutive bytes of standard information and an additional number of bytes of information specific to the disk driver.

logical end-of-file: The position 1 byte past the last byte in a file; equal to the actual number of bytes in the file. Compare **physical end-of-file.**

`long`: A 32-bit integer data type whose range is –2,147,483,648 to 2,147,483,647. Identical to type `int` in A/UX.

`long double`: In ANSI C, a floating-point type that may, but need not, have more precision and range than the `double` type. In A/UX C (but not all ANSI Cs), `long double` is synonymous with `double`.

Macintosh interfaces: A set of **interfaces** that enable you to access *A/UX Toolbox* routines (User Interface Toolbox and Macintosh Operating System) from A/UX C.

Macintosh Programmer's Workshop (MPW): Apple's software development environment for the Macintosh family.

macro: Text that is replaced by another string of text at compile time.

`main`: The name of the function that is the entry point for every C program.

Main: The default segment name.

main segment: The segment containing the main program.

manager: A set of data structures and routines that perform related Toolbox or Macintosh Operating System functions. For instance, the Window Manager handles the display and manipulation of windows on the Macintosh screen.

MC68020: The microprocessor in the Macintosh II computer.

MC68030: The microprocessor in the Macintosh IIx computer and later models. It supports the MC68020 instruction set and contains a subset of the Paged Memory Management Unit.

MC68851: The Motorola 68851 Paged Memory Management Unit (PMMU), an optional integrated circuit that provides full memory mapping in the Macintosh II, including 24- to 32-bit address mapping and virtual memory support.

MC688881: The floating-point coprocessor in the Macintosh II computer.

MC688882: The floating-point coprocessor in the Macintosh IIx computer. It supports the MC68881 instruction set.

memory block: An area of contiguous memory within a heap zone.

MPW: See **Macintosh Programmer's Workshop.**

newline (\n): A control code that advances the print position or cursor to the left margin of the next output line.

newline character: Any character, but usually carriage return (ASCII code 13), that indicates the end of a sequence of bytes.

newline mode: A mode of reading data where the end of the data is indicated by a newline character (and not by a specific byte count).

normalized number: A floating-point number that can be represented with a leading significand bit of 1.

object: An area of memory that can be examined and stored into. It has an **identifier** and an **address**.

object file: A file containing specifications for data and code-module contents, references to other code and data modules, and segmentation. Object files are produced by the assembler and compiler and are passed as input to the linker.

obsolescent: A feature that may now work, but will not work on future compilers. Using an obsolescent feature usually produces a compiler warning.

obsolete: A feature that worked on earlier compilers, but is not supported on the current compiler. Using an obsolete feature usually produces a compiler error, but it may silently produce erroneous code.

open file: A file with an access path. Open files can be read from and written to.

Operating System: The lowest-level software in the Macintosh. It does basic tasks such as I/O, memory management, and interrupt handling.

output driver: A device driver that receives data via a serial port and transfers it to an application.

package: A set of routines and data types that is stored as a resource and brought into memory only when needed.

Paged Memory Management Unit (PMMU): The Motorola 68851 Paged Memory Management Unit, an optional integrated circuit that provides full memory mapping in the Macintosh II, including 24- to 32-bit address mapping and virtual memory support.

parameter: An input to a routine.

parameter block: A data structure used to transfer information between applications and certain Macintosh Operating System routines.

partial pathname: A pathname beginning from any directory other than the root directory. A partial pathname either contains no colons or has a leading colon.

Pascal-style function: A function, written in Pascal, C, or assembly language, that is declared in C using the `pascal` specifier.

Pascal string: A sequence of characters that begins with a length byte and has a maximum length of 255 characters. This format is used by the Pascal compiler for variables of type `STRING`, and is the default form of string created by the assembler.

pathname: A series of concatenated directory names and filenames that identifies a given file or directory. See also **partial pathname** and **full pathname.**

path reference number: A number that uniquely identifies an individual access path; assigned when the access path is created.

physical end-of-file: The position 1e byte past the last allocation block of a file; equal to 1 more than the maximum number of bytes the file can contain. Compare **logical end-of-file.**

physical size: The actual number of bytes a memory block occupies within its heap zone.

PMMU: The Motorola 68851 Paged Memory Management Unit, an optional integrated circuit that provides full memory mapping in the Macintosh II, including 24- to 32-bit address mapping and virtual memory support.

pointer: The address of an object. A pointer and an `int` are the same size.

pragma: An implementation-dependent compiler directive introduced by the ANSI-C keyword `#pragma`. By definition, pragmas are not portable.

preprocessor: Part of the C compiler that provides file inclusion and macro substitution.

predefined symbol (also called **preprocessor symbol**): One of a set of constants defined to be 1, equivalent to writing "`#define symbol 1`" at the beginning of the source file.

read/write permission: Information associated with an access path that indicates whether the file can be read from, written to, both read from and written to, or whatever the file's open permission allows.

register-based routine: A Toolbox or Macintosh Operating System routine that receives its parameters and returns its results, if any, in registers.

register variable: An **automatic variable** that is allocated to a register. The `register` specifier tells the compiler to allocate a register to a variable if possible.

regular expressions: A language for specifying text patterns, using a special set of metacharacters.

relocatable block: A block that can be moved within the heap during compaction.

relocatable value: an expression or identifier whose value is relative to the start of a particular segment. The memory location represented by such an expression cannot be known at assembly time, but the relative values of two such expressions (that is, the difference between them) can be known if they refer to the same segment. Compare **absolute value**.

resource file: Common usage for the **resource fork** of a Macintosh file.

resource fork: The part of a file that contains data used by an application, such as menus, fonts, and icons. An executable file's code is also stored in the resource fork.

result: An output from a routine.

root directory: The directory at the base of a file catalog.

routine: The XXX.

scalar: A pointer or object of arithmetic type.

script: Same as **command file.**

segment: One of several parts into which the code of an application can be divided. Not all segments need to be in memory at the same time. Segment names are specified at compile time by means of the `-s` compiler option or the `#pragma segment` *segment-name* preprocessor directive.

server: A node that manages access to a peripheral device.

`short`: A 16-bit integer data type whose range is –32,768 to 32,767.

side effect: Any operation that accesses a volitile object, modifies an object, modifies a file, or calls a function that does any of these operations.

signal: A software interrupt that causes a program to be diverted from its normal execution sequence.

span-dependent optimization: The XXX.

stack-based routine: A Toolbox or Macintosh Operating System routine that receives its parameters and returns its results, if any, on the stack.

Standard C Library: A library of constants, data types, and functions that support both formatted and low-level I/O, string manipulation, character classification, memory allocation, and mathematical functions. The A/UX C Standard C Library has all the functions required by the ANSI C standard, as well as some Apple extensions.

standard error: Same as **diagnostic output.**

standard input: The file from which A/UX tools generally read input if no filename parameters are specified. Standard input is by default the text entered while the tool is executing, and can be redirected to other files, windows, and selections. In C, standard input is referenced using stream `stdin`.

standard output: The file to which many A/UX tools write their output. Standard output appears following the commands being executed in the active window by default, and can be redirected to other files, windows, and selections. In C, diagnostic output is referenced using stream `stdout`.

`Str255`: In A/UX C, an array of type `unsigned char`.

stream: A file with associated buffering.

`struct`: A record data type.

subdirectory: Any directory other than the root directory.

system call: A request to execute a named operating-system function; also, the name of the function itself.

system error ID: An ID number that appears in a system error alert to identify the error.

system space: A reserved memory region that only processes belonging to the root user can access.

`__TIME__`: A reserved preprocessor symbol that represents the current time.

token: The minimal lexical element used in a C compiler. The catagories of tokens are keywords, identifiers, constants, string literals, operators, and puncuators.

tool: A program that runs under A/UX.

Toolbox: Same as **User Interface Toolbox.**

Transcendental Functions Package: A Macintosh package that contains trigonometric, logarithmic, exponential, and financial functions, as well as a random number generator.

trap: An exception caused by instruction execution. It arises from process recognition of abnormal conditions during instruction execution or from use of the specific instruction whose normal behavior is to cause an exception.

type: A kind of memory object characterized by certain storage properties.

type qualifier: A keyword that can modify a type and give it certain charicteristics, such as permanance or impermanence.

unbuffered: A buffer style that does not use a buffer for I/O; reading and writing is done one character at a time.

undefined external: an identifier whose value is defined external to the containing object file.

unlock: To allow a relocatable block to be moved during heap compaction.

unmounted volume: A volume that hasn't been inserted into a disk drive and had descriptive information read from it, or a volume that was previously mounted and has since had the memory used by it released.

unpurgeable block: A relocatable block that can't be purged from the heap.

`unsigned char`: An 8-bit character data type whose range is 0 to 255.

`unsigned int`: A 32-bit integer data type whose range is 0 to 4,294,967,295. Identical to `unsigned long`.

`unsigned long`: A 32-bit integer data type whose range is 0 to 4,294,967,295. Identical to `unsigned int`.

`unsigned short`: A 16-bit integer data type whose range is 0 to 65,535.

User Interface Toolbox: The software in the Macintosh ROM that helps you implement the standard Macintosh user interface in your application.

user space: A memory region in an A/UX system to which a user process has access.

`void`: A data type used to declare functions that take no parameters or return no value. The `void` type can be used to cast expressions where values are not used. Pointers to `void` are also allowed.

volume: A piece of storage medium formatted to contain files; usually a disk or part of a disk. A 3.5-inch Macintosh disk is one volume.

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft Word software. Final pages were created on Apple LaserWriter® printers. Line art was created using Adobe Illustrator. PostScript®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of ITC Garamond®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writer: Tom Berry
Developmental Editor: Lorraine Aochi
Illustrator: Barbara Smyth
Production Supervisor: Tess Lujan

Special thanks...
Additional thanks ...