Fortune C Language Guide
Ordering Fortune C Language

Order Numbers: 1000834-01 for the guide with disks
               1000835-01 for the guide without disks
Consult an authorized Fortune Systems dealer for copies of manuals and technical information.
Copyright © 1982 Fortune Systems Corporation. All rights reserved.

No part of this document may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent in writing from Fortune Systems Corporation. The information in this manual may be used only under the terms and conditions of separate Fortune Systems Corporation license agreements.

Printed in U.S.A
1 2 3 4 5 6 7 8 9 0

Unix is a trademark of Bell Laboratories

Ordering Fortune C Language Guide

Order Number: 1000837-01
Please do not order products from the address shown below. Consult an authorized Fortune Systems dealer for copies of manuals and technical information.

Customer Comments

Your ideas about Fortune products and evaluations of Fortune manuals will be appreciated. Submit your comments to the Publications Department, Fortune Systems Corporation, 300 Harbor Boulevard, Belmont, CA 94002. By submitting any idea, evaluation, or other information to Fortune Systems Corporation, you consent to any use or distribution of such information deemed appropriate by Fortune Systems Corporation. Fortune Systems Corporation shall have no obligation whatsoever with respect to such information.

Disclaimer of Warranty and Liability

No representations or warranties, expressed or implied, of any kind are made by or with respect to anything in this manual. By way of example, but not limitation, no representations or warranties of merchantability or fitness for any particular purpose are made by or with respect to anything in this manual.

In no event shall Fortune Systems Corporation be liable for any incidental, indirect, special or consequential damages whatsoever (including but not limited to lost profits) arising out of or related to this manual or any use thereof even if Fortune Systems Corporation has been advised, knew or should have known of the possibility of such damages. Fortune Systems Corporation shall not be held to any liability with respect to any claim on account of, or arising from, the manual or any use thereof.

For full details of the terms and conditions for using Fortune software, please refer to the Fortune Systems Corporation Customer Software License Agreement.
How to Use This Guide

The Fortune C Language Guide is designed to help you use the C language on the Fortune 32:16. The information included covers those aspects particular to the Fortune system in addition to helpful utilities and more advanced features for very experienced programmers.

The guide is not intended to teach you to program in C. Use the guide along with the C programming manual of your choice. Below is a list of recommended manuals.

- B. W. Kernighan, *Programming in C-A Tutorial*
- D. M. Ritchie, *C Reference Manual*

This package includes the following items:

- Fortune C Language Guide
- Two Master disks
- Fortune Systems software registration card

If any item is missing, contact your Fortune Systems dealer for a replacement.
## Contents

**How to Use This Guide** iv

### Section 1  Fortune C Language Guide

#### Part 1  Using C on the Fortune 32:16  1

- Starting Up Your Fortune 32:16  2
- Installing the C Compiler  5
- Formatting Disks  9
- Copying Your Disks  11
- Selecting and Leaving C  13
- The C Compiler (cc)  15

#### Part 2  The Fortune Symbolic Debugger  20

- Descriptions of the Debugger  21
- Definitions  24
- Organization  26
- Command Syntax  27
- Command Classification  30
- Commands to Set Up Environment/Display Information  31
- Commands for Source File Examination  38
- Commands for Controlling Execution  41
- Special Notes  46
- Operating Procedure  47

#### Part 3  Support Utilities, Libraries, and Machine Specific Aspects  49

- Archive  ar  50
- Link Editor  ld  53
- Make  58
Part 4 The Fortran/C/Language Interface 77
   Procedure Names 77
   Data Representation 77
   Return Values 78
   Argument Lists 79

Section 2 Library Routines 1

Part 1 System Routines 3

Part 2 Library Functions 1

Part 3 File Formats 1
SECTION I
FORTUNE C LANGUAGE GUIDE

This section contains documentation on the installation of C on the Fortune 32:16, the C compiler, the Fortune symbolic debugger, and machine specific aspects of the Fortune C language. In addition, the support utilities and the interface between Fortran and C on the Fortune 32:16 are discussed.
C is a general-purpose programming language that runs under the Fortune Operating System on the Fortune 32:16. The Fortune Operating System is a modified version of UNIX, an operating system developed by Bell Laboratories. The C language is simple, efficient, and appropriate for a wide variety of programming applications.

In this section you'll learn first to power up your system. You'll also learn:

- How to install C
- How to format and copy disks
- How to select and leave C
- How to use the C compiler
Starting Up Your Fortune 32:16

The first step is to plug in the system. Do this with the power on/off switch in the off position. For your safety, and the protection of the system, use a three-pronged electrical outlet that fits the connector.

Now push in the white dot on the switch to set the power switch to on. Test the airflow with your hand to make sure that the fan is operating.

First you will see the cursor blinking on the screen. Then the message "Fortune Systems 32:16 Please Wait," with the "Please wait" blinking appears. When you see the heading "Please enter the current date and time," the system is ready to receive information.

Use the following procedure to log onto your system. It is read from left to right. The system category shows you what will appear on the screen. Type what you see in the user category. The comments column provides useful information.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Please set the current date and time, then press (RETURN):</td>
</tr>
<tr>
<td></td>
<td>Today's date is: mm/dd/yy</td>
</tr>
<tr>
<td>User</td>
<td>(current date) (RETURN)</td>
</tr>
<tr>
<td></td>
<td>Type in six digits to represent month, date, and year or press the Return key to accept the date displayed. You don't have to type the slashes.</td>
</tr>
<tr>
<td>Procedure</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>System</td>
<td>Use the Back Space key to backup within a line and the ↑ and ↓ keys to move up and down between date and time. The Cancel key bypasses this entry altogether.</td>
</tr>
<tr>
<td>System</td>
<td>Type in four digits to represent hours and minutes or press the Return key to accept the time displayed. You don't have to type the colon.</td>
</tr>
<tr>
<td>User</td>
<td>Type y or n to indicate correct/incorrect date and time. Typing n returns you to beginning of date.</td>
</tr>
<tr>
<td>User</td>
<td>The dots blink while the system checks the files. If any other message appears get help.</td>
</tr>
</tbody>
</table>

FORTUNE SYSTEMS 32:16
Press **HELP** For Assistance
Type in your name and press **RETURN**
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
</table>
| User      | (account name) **(RETURN)**  
Type your account name. Type newuser to create a new account. |
| System    | Type in your password and press **(RETURN)**: |
| User      | (password) **(RETURN)**  
Type your password. This is requested only if you have one assigned. |
| System    | _% of the available space is currently in use._  
When your system reaches 90% full, archive some files to free up more work space. |
| FORTUNE SYSTEMS GLOBAL MENU | Make your selection from the Global Menu. |
Installing the C Compiler

Before you begin to use C, you need to install the programs and files from the two master flexible disks to a hard disk on the system you are using. You need a minimum of 384k of memory to load the C compiler.

The C compiler is loaded through the product maintenance menu. When the global menu appears, select Product Maintenance.

Follow the procedures below to load the software. To do this procedure you must be logged in as manager. First shut down the system. Then turn it on again while holding down the Cancel key.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System FORTUNE SYSTEMS GLOBAL MENU</td>
<td>To power down, select s2.</td>
</tr>
<tr>
<td>User s2</td>
<td></td>
</tr>
<tr>
<td>System System Management</td>
<td>Choose 30 from the system</td>
</tr>
<tr>
<td>User 30</td>
<td>management menu.</td>
</tr>
<tr>
<td>System Fortune Systems 32:16 ShutDown</td>
<td>(takes about 30 seconds)</td>
</tr>
<tr>
<td></td>
<td>Do you want to continue?</td>
</tr>
<tr>
<td>User yes</td>
<td>Wait for system messages.</td>
</tr>
<tr>
<td>Procedure</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Software shutdown starting, please wait. Software shut down complete Hardware shut down starting, please wait Hardware shut complete Please turn the Fortune Systems 32:16 off</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>Press off switch. Now turn on system again, holding down CANCEL.</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Type any highlighted key. Press the F7 key.</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>F7</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Set boot file name</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>hd02/sa/reconf RETURN EXECUTE Move cursor to Max process size.</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Max process size</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>256 RETURN F1 F4</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Today's date is Current time is: Bypass this.</td>
</tr>
<tr>
<td>Procedure</td>
<td>Comments</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>(RETURN)</td>
</tr>
<tr>
<td></td>
<td>(RETURN)</td>
</tr>
<tr>
<td>yes</td>
<td>(RETURN)</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Type in your name and press (RETURN)</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>(your account name)  (RETURN)</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>FORTUNE SYSTEMS GLOBAL MENU</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>s5  (RETURN)</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>PRODUCT MAINTENANCE</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>i  (RETURN)</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Fortune Systems Corporation Product Maintenance Please insert flexible disk volume 1. Press (RETURN)</td>
</tr>
<tr>
<td></td>
<td>Put the disk labelled &quot;development set&quot; in the drive.</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>(RETURN)</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>This flexible disk is labeled: Development Set xxxx Volume 1 (date)</td>
</tr>
<tr>
<td>Procedure</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td><strong>System</strong> Proceed with installation?</td>
<td></td>
</tr>
<tr>
<td>(y/n):</td>
<td></td>
</tr>
<tr>
<td><strong>User</strong> y (RETURN)</td>
<td>The system puts a copy on the hard disk.</td>
</tr>
<tr>
<td><strong>System</strong> Copy phase of Development Set ...</td>
<td></td>
</tr>
<tr>
<td>Menu update ...</td>
<td></td>
</tr>
<tr>
<td>Development Set installation successfully</td>
<td></td>
</tr>
<tr>
<td>completed.</td>
<td></td>
</tr>
<tr>
<td>Press (RETURN) for menu or select ahead</td>
<td></td>
</tr>
<tr>
<td><strong>User</strong> (RETURN)</td>
<td>Remove the flexible disk.</td>
</tr>
<tr>
<td></td>
<td>Repeat the process to install the second disk</td>
</tr>
<tr>
<td></td>
<td>labelled &quot;C Compiler.&quot;</td>
</tr>
<tr>
<td><strong>System</strong> FORTUNE SYSTEMS GLOBAL MENU</td>
<td>You're at the global menu.</td>
</tr>
</tbody>
</table>
Formatting Disks

Before you can use a blank flexible disk to store your application or other files, the disk must be formatted. From the global menu use this procedure to format a flexible disk.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System</strong> GLOBAL/MENU</td>
<td></td>
</tr>
<tr>
<td>S1 System Utilities</td>
<td></td>
</tr>
<tr>
<td><strong>User</strong></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>(RETURN)</td>
</tr>
<tr>
<td><strong>System</strong> SYSTEM UTILITIES MENU</td>
<td></td>
</tr>
<tr>
<td><strong>User</strong></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>(RETURN)</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td></td>
</tr>
<tr>
<td>FORMAT FLEXIBLE DISK</td>
<td></td>
</tr>
<tr>
<td>Do you want to continue (yes or not)?:</td>
<td></td>
</tr>
<tr>
<td><strong>User</strong></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>(RETURN)</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td></td>
</tr>
<tr>
<td>Please wait for completion message</td>
<td></td>
</tr>
<tr>
<td>Do not press any key while this message is on the screen.</td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Your request is complete Please Remove Your Flexible Disk</td>
<td>Your disk is formatted. Remove the disk.</td>
</tr>
<tr>
<td>-Press (RETURN) for menu or select ahead</td>
<td></td>
</tr>
</tbody>
</table>

**User**

**System** FORMAT FLEXIBLE DISK

You can repeat the formatting procedure at this point by beginning at step 3 again.

**User**

**System** SYSTEM UTILITIES MENU

**User**

**System** FORTUNE SYSTEMS GLOBAL MENU

10
Copy your disks

Make a copy of your software as soon as possible. The procedure below is used to back up your master disks. To do this procedure, you must be logged in as manager.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>FORTUNE SYSTEMS GLOBAL MENU</td>
</tr>
<tr>
<td>User</td>
<td>s5 (RETURN)</td>
</tr>
<tr>
<td>System</td>
<td>PRODUCT MAINTENANCE</td>
</tr>
<tr>
<td>User</td>
<td>b (RETURN)</td>
</tr>
<tr>
<td>System</td>
<td>PRODUCT SELECTION MENU</td>
</tr>
<tr>
<td>User</td>
<td>cc</td>
</tr>
<tr>
<td>System</td>
<td>Fortune Systems Corporation Product Maintenance</td>
</tr>
<tr>
<td>User</td>
<td>y (RETURN)</td>
</tr>
<tr>
<td>Procedure</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>System</td>
<td>Please label a blank flexible disk:</td>
</tr>
<tr>
<td>'C' compiler</td>
<td></td>
</tr>
<tr>
<td>1000837-01</td>
<td></td>
</tr>
<tr>
<td>Volume 1</td>
<td></td>
</tr>
<tr>
<td>(date)</td>
<td></td>
</tr>
<tr>
<td>Insert the disk into drive #0.</td>
<td>Be sure the disk was previously formatted.</td>
</tr>
</tbody>
</table>

User

System

Copy phase ...

Successfully ...

-Press (RETURN) for menu or select ahead

User

System

FORTUNE SYSTEMS GLOBAL MENU

Repeat the process to back up the development set disk, choosing ds.
Selecting and Leaving C

From the global menu use the following procedure to choose the UNIX command interpreter where you will run C.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System FORTUNE SYSTEMS GLOBAL MENU</td>
<td></td>
</tr>
<tr>
<td>User !sh (RETURN)</td>
<td>Type !sh. Use the shift 1 key for !. You are now in direct communication with the operating system.</td>
</tr>
<tr>
<td>System $</td>
<td>The $ shows that the system is ready.</td>
</tr>
</tbody>
</table>

Use the ed editor on your Fortune system to develop and edit programs. Refer to your Fortune Operating System Guide for information about using ed.

When you have finished your work use the following procedure to log out.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>System $</td>
<td></td>
</tr>
<tr>
<td>User (CTRL)d</td>
<td>Press the CTRL key and d at the same time.</td>
</tr>
<tr>
<td>Procedure</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| **System** | -Press RETURN for menu  
or select ahead |
| **User**  | (RETURN) | Pressing the Return key  
returns you to the global menu. |
The C Compiler (cc)

CC is the command that activates the C compiler. The compiler reads in source code, and translates that code into machine language which can be understood by the computer. The following are the five steps involved in compiling a C program.

1. Preprocessor
   This processes any # sign statements.

2. C Compiler
   Code in filename.c is translated into assembly language.

3. Optimizer
   Code is optimized and thereby reduced in size. The optimizer also increases the runtime speed.

4. Assembler
   The object file (filename.o) is created.

5. Load
   The executable file (a.out) is created.

A set of options, described in the next few pages, provides variations in compiling results.

To compile a C program enter

    cc (options) filename.c ...

The argument, or filename you enter whose name ends with .c is a C source program. It is compiled and an executable file named a.out is created. In addition, a .o file is created if the -c option is used or more than one c source file is compiled with the same command. This is the object file, the compiled C program. The .o file can later be processed by the loader, then executed. For example, for the file named test.c:
You Enter

cc test.c

Results

a.out

Any number of .c files may be compiled into one a.out file. Again, .o files will also be created for each .c file.

You Enter

cc test1.c test2.c

Arguments other than the C options described below are taken to be loader option arguments or C-compatible object programs. These object programs are typically produced by an earlier cc run, or libraries of C-compatible routines. These programs and the results of any specified compilations are loaded (in the named order) to produce a runnable program named a.out. To create only .o files, use the -c option. No a.out file will result.

You Enter

cc -c test1.c

Object files (.o) may be linked to create an a.out file.

You Enter

cc test1.o test2.o

Already compiled files (.o) and .c files may be run through the compiler with the following results.

You Enter

cc test1.o test2.c

Results

a.out test2.o
Using the -o option you can name an a.out file.

You Enter                      Results

cc -o test test1.o test2.c      test
                                     test2.o

The following options are interpreted by cc.

-c  Does not link object file with libraries. Leaves only the .o file.

You Enter                      Results

cc -c test.c                    test.o

-o  Calls an object code optimizer. Code size will be reduced 20-25% in size and result in a faster running file.

-v  Verbose. Compiler lists passes on the screen as they are executed.

You Enter                      Results

cc -v test.c                    /usr/lib/cpp -DMC68000 -Uvax
                                  file.c /tmp/ctm0013H.s
                                  /usr/lib/ccom /tmp/ctm001314.s
                                  /tmp/ctm00/313.s
                                  /usr/lib/ac -o test.o /tmp/ctm001313.s
                                  /usr/bin/ld /usr/lib/crt.o file.o
                                  /usr/lib/libc.a
-G The stack growth checking is turned off. This improves the code slightly as long as the stack is not used extensively. It decreases the text size.

Do not run -G on a program that allocates more than 8K of stack.

For example, the following program will fail under the -G option.

```c
main ( )
{
    int x [4000], i;
    For (i = 0; i < 4000; i++)
        x [i] = 0;
}
```

-E Runs only the macro preprocessor on the named C programs. The result is sent to the screen.

<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results (on the screen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -E test.c</td>
<td>#1 &quot;test.c&quot;</td>
</tr>
<tr>
<td></td>
<td>(text of program)</td>
</tr>
</tbody>
</table>

-C This prevents the macro preprocessor from removing comments.

<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -E test.c</td>
<td>Comments are removed</td>
</tr>
<tr>
<td>cc -E -C test.c</td>
<td>Comments remain</td>
</tr>
<tr>
<td>cc -E -C test.c ff.c</td>
<td>Comments are put into ff.c</td>
</tr>
</tbody>
</table>

-0 output Names the final executable file output and leaves the a.out file undisturbed.
-Dname=def Defines the name to the preprocessor, as with
-Dname #define. If you give no definition, the name is
defined as one.

You Enter                              Results

cc -DFLEXNAMES                       FLEXNAMES is defined and assigned
                                             the value 1.

cc -DFLEXNAMES=12                     FLEXNAMES is defined and assigned
                                             the value 12.

-Uname Removes any initial definition of name in the preprocessor.

-Idir # include files whose names do not begin with / are always
       looked for first in the directory of the file argument,
       then in directories named in -I options, then in
       directories on a standard list.
The Fortune Symbolic Debugger

Included on the master disks for C is the Fortune symbolic debugger. Fortune Systems Symbolic Debugger (fdb) is a high-level debugging tool developed by the Fortune Systems Corporation. Fdb is language independent so it will serve as a common debugger for all the high level (compiler) languages supported on the Fortune system.
Description of the Debugger

Fdb is a symbolic debugger which can be used with the C language. The format of the fdb command is:

    fdb [objfil [directory]]

You use it to examine your files and to provide a controlled environment for file execution. Objfil is an executable program file which has been compiled with the -g (debug) option. The default for objfil is a.out. Core file is not utilized. Directory is the working directory.

It is useful to know that at any time there is a current line and current file. The default for the current file is the file debugged. However, the current file may be changed with the source file examination commands. There are two types of current line. One is current print line, and the other is current execute line. The current execute line can only be changed with program execution while the current print line can be changed with file examination commands.

Names of variables are written just as they are in C. Variables local to a procedure may be accessed using the form 'procedure: variable'. If no procedure name is given, the procedure containing the current line is used by default. It is also possible to refer to structure members as 'variable.member', pointers to structure members as 'variable->member' and array elements as 'variable [number]' and array elements. Combinations of these forms may also be used.

FILES

The file used by fdb is a.out.
DIAGNOSTICS

Error reports are self-explanatory.

BUGS

Error checking for structured variable elements are not performed.

The fdb commands are summarized below.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Exits the shell (escape)</td>
</tr>
<tr>
<td>,</td>
<td>Displays the content of a variable (same as display command)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Displays the address of a variable</td>
</tr>
<tr>
<td>RETURN key</td>
<td>Repeats the previously executed command</td>
</tr>
<tr>
<td>alias</td>
<td>Defines or cancel alias</td>
</tr>
<tr>
<td>break</td>
<td>Sets up a breakpoint</td>
</tr>
<tr>
<td>comment</td>
<td>Allows a comment line</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes breakpoint(s)</td>
</tr>
<tr>
<td>display</td>
<td>Displays the content of a variable (same as , command)</td>
</tr>
<tr>
<td>dump</td>
<td>Dumps memory contents</td>
</tr>
<tr>
<td>equate</td>
<td>Defines or cancels replacement string</td>
</tr>
<tr>
<td>file</td>
<td>Redirects source/input/output file</td>
</tr>
<tr>
<td>Command</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Searches a given string from the source file</td>
</tr>
<tr>
<td><code>go</code></td>
<td>Starts or resumes debugged program execution</td>
</tr>
<tr>
<td><code>help</code></td>
<td>Shows the summary of fdb commands</td>
</tr>
<tr>
<td><code>print</code></td>
<td>Prints source lines</td>
</tr>
<tr>
<td><code>quit</code></td>
<td>Exits from fdb and return to shell</td>
</tr>
<tr>
<td><code>restart</code></td>
<td>Restarts the debug session with optional parameter</td>
</tr>
<tr>
<td><code>set</code></td>
<td>Sets debugger options such as user definable prompt string</td>
</tr>
<tr>
<td><code>show</code></td>
<td>Shows status of debug session such as breakpoint, file, window, alias, last command, equate and procedure</td>
</tr>
<tr>
<td><code>trace</code></td>
<td>Traces program execution</td>
</tr>
<tr>
<td><code>walk</code></td>
<td>Single step execution</td>
</tr>
</tbody>
</table>
Definitions

The following terms are defined as used in this description of fdb.

**Breakpoint**
A location in a program's execution at which either some debugging command is to be performed or the user wishes to gain control.

**Command**
Debugging command

**Debug option**
A compiler directive to have extra Symbol table entries added which are utilized by fdb. The option is specified by -g, thus, sometimes it is called -g option under UNIX environment.

**Debugging command**
A directive that controls the behavior of a debugger.

**Debugging session**
A period of time during which a debugger is used.

**Debugging mode**
Execution of a program in conjunction with a debugger.

**Linker/loader**
The function of a linker is to link the object modules and produce an executable load module. The function of a loader is to load the load module from disk into memory. The linker is called the loader and the loader is the kernel in UNIX.
Object/load module
The input to the linker is called the object module and the output is the load module. There is no clear difference between object and load module in UNIX. Thus the term object and load modules are used interchangeably.

Symbolic debugging
The debugging of programs in terms of their source level names and constructs.

Trace
A display of the dynamic activity of some aspect of a program. Fdb supports the execution trace, the procedure trace and the variable trace.
Organization

The following figure shows how fdb is utilized in program execution.
Command Syntax

The following are the general rules of the fdb command.

CASE RULE

There is no difference between upper and lower case letters. Combinations of upper and lower case letters are allowed. This rule also applies to the fdb keywords. For example, the following commands are equivalent:

- equate
- EQUATE
- EquAte

Upper and lower case letters may be distinct in variable and procedure names. This is language dependent.

3-CHARACTER RULE

Every command can be abbreviated to three characters if desired. For example, the following strings are all legal commands.

- BRE for break
- DEL for delete
- EQU for equite ... etc.

Some commands may even be abbreviated to one character (please see HELP for details). However, if a command is spelled with more
than the allowed abbreviation (one or three characters), the whole command string should be spelled out. For example, EQ, EQUA and EQUAT are illegal while E and EQU are legal.

LEADING BLANK RULE

All the leading blanks in a command are ignored. So, the following commands are equivalent. One or more blanks and tab characters are equivalent to one blank character.

EQUATE
eQU
Equate

MULTIPLE COMMANDS PER LINE

Multiple commands per line are allowed if they are separated by the semicolon (;). Thus, a semicolon before the Return key has its own meaning (please see NEWLINE for details).

Note that each command in a multiple command line is interpreted. So the first command is performed regardless of the error condition in the subsequent commands. For example:

command 1; command 2; command 3 (RETURN)

is equivalent to

command 1 (RETURN)
command 2 (RETURN)
command 3 (RETURN)

This rule does not apply when a semicolon appears in a string, in a COMMENT command, or in a BREAK - DO command. For example, each of the following commands is a single command.
Command Classification

This section describes all the debugging commands supported by fdb. Commands are classified into three categories:

- Commands to set up environment/display information
- Commands for source file examination
- Commands for controlling execution

Each command is presented with the command's grammar in Backus-Naur form, a functional description of the command, and examples.
Commands to Set Up Environment/Display Information

DISPLAY

The following are the commands used to set up display variables. In BNF notation, display variable is defined as:

<display var> ::= ( , | DISPLAY ) <procedure><variable><format spec>;
<precedure> ::= <empty> | <procedure name> : ;
<format spec> ::= <empty> | <int spec>:<float spec>:<char spec>;
<int spec> ::= <byte size> <int form>;
<byte size> ::= <empty> | b! h! l ;
<int form> ::= x| d| o| u ;
<float spec> ::= f| g ;
<char spec> ::= c | <string form> ;
<string form> ::= <string size> s ;
<string size> ::= <empty> | <unsigned> ;

This command displays the values of variable(s) at program suspension. The values are displayed according to the user format specification. If format specification is omitted, variables are formatted according to their data type as declared in the program.

For example, suppose the types and contents of variables i, p, a and j are defined as follows:

<table>
<thead>
<tr>
<th>variable type</th>
<th>name</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>i</td>
<td>'x'</td>
</tr>
<tr>
<td>char</td>
<td>*p</td>
<td>&quot;abcxy&quot;</td>
</tr>
<tr>
<td>char</td>
<td>a[3]</td>
<td>&quot;ABC&quot;</td>
</tr>
<tr>
<td>int</td>
<td>j</td>
<td>0x12345678</td>
</tr>
</tbody>
</table>

The fdb commands and its output values for the example are:

,i : x
,i/x : 0x78000000
In BNF notation, equate is defined as:

\[
\text{<equate> ::= EQUATE <alpha> ( <empty> : <string> )}
\]

The EQUATE command equates a character to a data string. For the equated character to be expanded, an escape character (%) should precede the equated character. When the equated character appears in a command, it will be expanded inline prior to executing the command. Thus, equate could be used to combine the multiple commands into one or alias commands.

An equate command may be cancelled by equating the previously defined character to a null (empty) string.

Fdb will detect and report recursive equate definitions. For example:

Equate to long variable name

equ a "employee" : define a as an equated character to "employee"
display %e.ar : display the content of variable employer
display %e.name : display the content of variable employee.name
equ a , %e.ssn : cancel the equate definition
,m %e.ssn : might have been employee.ssn but illegal since equation a was cancelled
Equate to Multiple commands

equ b " SHOW ARG; SHOW LAN; SHOW EQU; ! who "
%b : shows the arguments defined, source language, equated characters and the name logged on the system

Equate to user defined command (in this case ALIAS is better than EQUATE)

equ w "PRINT .-5 ! 11"
%w : print 5 lines before and after the current line

HELP

The HELP command lists every fdb command with a short description. Help can be invoked by pressing the HELP key, typing help, or typing ?. A command that can be abbreviated to one character is represented by one lower case character in parentheses. The following is a list of commands and their descriptions on the help facility.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>: shell escape</td>
</tr>
<tr>
<td>,</td>
<td>: display the content of a variable</td>
</tr>
<tr>
<td>&amp;</td>
<td>: display the address of variable</td>
</tr>
<tr>
<td>RETURN key</td>
<td>: repeat previous command</td>
</tr>
<tr>
<td>ALIAS</td>
<td>: define/cancel alias</td>
</tr>
<tr>
<td>BREAK(b)</td>
<td>: set up a breakpoint</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>COMMENT</td>
<td>:comment line</td>
</tr>
<tr>
<td>DELETE(d)</td>
<td>:delete breakpoint(s)</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>:display the content of variable (same as ,)</td>
</tr>
<tr>
<td>DUMP</td>
<td>:dump memory contents</td>
</tr>
<tr>
<td>EQUATE(e)</td>
<td>:define/cancel replacement string</td>
</tr>
<tr>
<td>FIND(f)</td>
<td>:search a given string from the source file</td>
</tr>
<tr>
<td>FILE</td>
<td>:redirect source/input/output files</td>
</tr>
<tr>
<td>GO(g)</td>
<td>:start or resume execution</td>
</tr>
<tr>
<td>HELP(h)</td>
<td>:shows legal fdb commands</td>
</tr>
<tr>
<td>PRINT(p)</td>
<td>:print source lines</td>
</tr>
<tr>
<td>QUIT(q)</td>
<td>:exit from fdb and return to shell</td>
</tr>
<tr>
<td>RESTART(r)</td>
<td>:restart the debug session with optional parameter</td>
</tr>
<tr>
<td>SET</td>
<td>:set debugger options</td>
</tr>
<tr>
<td>SHOW(s)</td>
<td>:show status for breakpt/argument/file/equate procedure</td>
</tr>
<tr>
<td>TRACE(x)</td>
<td>:trace program execution</td>
</tr>
<tr>
<td>WALK(w)</td>
<td>:single step execution</td>
</tr>
</tbody>
</table>

SHOW STATUS

In BNF notation, show status is defined as:

\[
<\text{show status}> ::= \text{SHOW} ( \text{BREAKPOINT} \mid \text{FILE} \mid \\
\text{WINDOW} \ <\text{unsigned}> \mid \\
\text{ALIAS} \mid \text{COMMAND} \mid \text{EQU} \mid \text{PROCEDURE} ) ;
\]

This command is used to show information about the current debugger session at the user's terminal. The information that could be displayed is:

- Breakpoints that are currently set
- Input/output/source files
- A few lines around the current line
- Alias definitions
- Last command as seen by fdb (expanded in case of alias)
- List of all equate symbols and their current definitions
- Procedure stack, for example, the procedure names called to reach the current stop point

These are examples of the SHOW STATUS command.

- \text{SHOW \ PROCEDURE}: procedure names in frame stack
- \text{SHOW \ BREAKPOINT}: show all the breakpoints defined
- \text{SHOW \ EQUATE}: show all the equate definitions
- \text{SHOW \ WINDOW 4}: print 9 lines around the current line
COMMENT

Fdb prints the comment line as entered on the output device. This command is used to document the debug session when fdb output is not standard output (terminal). For example:

    COMMENT next statement is to test error condition
    EQU a; COM ;; This line has two commands even if many ;'s appeared

ALIAS

In BNF notation, ALIAS is defined as:

<alias> ::= ALIAS (<alias define>:<alias cancel>) ;
<alias define> ::= <def string><replace string> ;
<alias cancel> ::= <def string> ;

This command allows a user to define his/her own debugger command. The user can rename existing fdb commands or combine a few commands into one at his/her convenience.

To redefine the already defined alias, a user should cancel it before redefine. A user can use SHOW ALIAS command to see the alias definitions.

If a semicolon is used in the alias replacement string, multiple commands alias, it must be enclosed in quotes. Note that Case Rule does not apply to the alias definition string. For example:

    ALIAS single WALK : redefine single as WALK
    ALIAS step "WALK; DISPLAY a"

To make fdb commands look like Unix Sdb (may not be recommended though), a user can set up alias definitions as follows:
VARIABLE ADDRESS

In BNF notation, variable address is defined as:

```
<var address> ::= &<variable> ;
```

This command is used to display the address of a variable. The address is always displayed in hexadecimal notation. For example:

- `&a` : address of variable a
- `&b [3]` : address of fourth element of array b

SET

In BNF notation, set is defined as:

```
<set option> ::= SET <debug option> ;
<debug option> ::= <user prompt> ;
<user prompt> ::= PROMPT (<EMPTY> | =) "<string>";
```

This command is used to set up a debug option. Currently only the user prompt setting is available. For example:

- `SET PROMPT = "+"` : debugger prompt is +
- `SET PROMPT  "Fortune fdb%"`
Commands for Source File Examination

Several commands are used in examining source files: file definition, find string, print source lines, and dump.

FILE DEFINITION

This is the BNF notation for file definition:

<file definition> ::= FILE <file name> { <file name>} ;
<file name> ::= (<empty> | < | >! > ) <identifier> ;

The file definition command is used to refine the source file or redirect the standard input and output devices. It is used to change the file specifications for debugger. Files for the debugged program can be redirected by run time arguments (see RESTART command).

When < or > is followed by a space, fdb will redirect input or output devices to standard devices. >> is to append to the end of existing file. For example:

FILE <profile> :execute fdb commands in profile
FILE /user/source/test.c :source file is /usr/source/test.c
FILE > .. /trace :save debug output in parent's directory
FILE > :print debug output on terminal

FIND STRING

In BNF notation, find is defined as:

<find> ::= FIND <string><range>
<range> ::= ( <single line> | <multiple line> | <count> ) ,<range> ;
The FIND command is used to search the source (current) file and print the source line(s) which contain the specified string.

The count is for the maximum number of lines to print, and the default values for the line number is the current line. For example:

FIND "Procedure" : search "Procedure" and print the first line that contains the string from the current line
FIND "if" 3 : find the first "if" from line number 3
FIND "count".-3!10 : find 10 occurrences of "count" from current-3 line
FIND "xyz" 10/100 : find "xyz" string from line #10 through 100

PRINT SOURCE LINES

In BNF notation, print source is defined as:

<print source> ::= PRINT <range> ;

The PRINT command is used to print the specified number of lines (count) from the given starting lines in the source. The default values for the starting line is the current line. For example:

PRINT : print the current line
PRINT .-10 11 : print (current -10) and current same as PRINT.-10, 11
PRINT .-10/ 11 : print (current-10) through line #11
PRINT .! 6 : print 6 lines from the current line
PRI 3, .-2/ .+3, 10 : print line #3, from (current-2) through (current+3) and line #10
DUMP

In BNF notation, dump is defined as:

\[
\text{<dump>} ::= \text{DUMP}<\text{dump option}> <\text{dump spec}> ; \\
\text{<dump option>} ::= <\text{empty}> | \text{C} | \text{X} ; \\
\text{<dump spec>} ::= <\text{range}> | <\text{var address}> ;
\]

This command is used to display the contents of memory. A user can display in character format or in hexadecimal. The default is in hexadecimal format.

Output format is:

Space designation: I for instruction space
D for data space

Memory address in hex
16 bytes of contents

The memory dump is displayed in a 16-byte unit, and the starting address is always a multiple of 16. If a dump is requested towards the end of a line, for example, mod(address) is between 13 and 15, two lines are displayed. For example:

DUMP 0x100 : dump between 0x100 and 0x10f
DUMP 3 : dump between 0x0 and 0xf
DUMP 0x100/0x200
DUMP NEXT : dump next 16 bytes
DUMP &a : dump the memory around var a
Commands for Controlling Execution

The following commands are used for controlling execution of the debugger: breakpoint, delete breakpoint, go, shell escape, walk, quit, trace, and restart.

BREAKPOINT

In BNF notation, breakpoint is defined as:

\[
\text{<breakpt definition>} ::= \text{BREAK } \text{<break loc> } \text{<break command>} ;
\]

\[
\text{<break loc>} ::= \text{<empty> } | \text{<static break> } ;
\]

\[
\text{<static break>} ::= \text{<module name> } \text{<statement spec>}
\]

\[
\text{<module name> ::= <empty> } | \text{<procedure name> } \{| \text{<procedure name> } \};
\]

\[
\text{<statement spec> ::= <line number>} ;
\]

\[
\text{<line number>} ::= \text{<integer>} ;
\]

\[
\text{<procedure name> ::= <identifier> ;}
\]

\[
\text{<break command>} ::= \text{<empty> } | \text{DO } \text{<fdb command>} ;
\]

This command causes a breakpoint to be set at the indicated line number in the source program. The program is stopped before the line is executed. If the specified line is not an executable statement such as a blank or comment line, the breakpoint is set to the first executable line after that.

The module name and/or line number may be omitted in which case the defaults are taken from the current procedure name and the current line number, respectively.

If break command is specified as DO - phrase, fdb executes the command(s) when the breakpoint is reached. Otherwise, the control is transferred to the user. For example:
Break : break at current line in the current procedure unconditionally
B SUB1: 4 : break at line #4 in the procedure SUB1
BREAK 10 DO ,a; ,b : break at line #10 and print the values of var a and b when the program stops

DELETE BREAKPOINT

In BNF notation, delete breakpoint is defined as:

<delete breakpt> ::= DELETE ( <empty> | ALL:<module name> <statement spec> ) ;

The DELETE command is used to remove the breakpoints. DEL ALL will delete all the breakpoints set up so far. If no parameter is given, then the breakpoint is deleted interactively. Each breakpoint location is printed and a line is read from the standard input. If the user response is d, del, y, yes or ok, then the breakpoint is deleted. Other responses are considered as no. For example:

Del GETCHAR: 4 : delete the breakpoint on line 4 of procedure GETCHAR
DELETE : no parameter, so interactive deletion
delete SUB1 3? no : user does not want to delete line #3 of SUB1
delete SUB3 10? ok : user wants to delete this breakpoint

Delete all : delete all the breakpoints

GO

In BNF notation, go is defined as:

<go> ::= GO ( <empty> | <statement number> ) ;
<statement number> ::= <unsigned> ;
The command causes the program to either start or resume execution. If a statement number is specified, the program execution is suspended after executing the specified number of lines from the current position.

The GO command is used to continue the program execution, ignoring the signal that caused the execution to stop (such as user interrupt).

The program will continue to execute until one of the following events occurs:

- Breakpoint
- Program error
- User interrupt
- Normal program exit

SHELL ESCAPE

In BNF notation, shell escape is defined as:

\[
<\text{shell escape}> ::= \! <\text{shell command}> ;
\]

This command allows the user to execute shell command in the middle of a debugging session. Shell allows multiple commands if separated by the semicolon. However, fdb uses the same convention. Therefore, multiple shell commands per line are not permitted in fdb. For example:

\begin{itemize}
  \item \texttt{!date} :print date and time on the input device
  \item \texttt{!date; !who} :multiple fdb commands
  \item \texttt{!date; who} :illegal, since multiple shell commands are not allowed
\end{itemize}

WALK

In BNF notation, walk is defined as:

\[
<walk> ::= \text{WALK} ( <\text{empty}> ; <\text{unsigned}> ) ( <\text{empty}> ; \text{IN} \; 1 ) ;
\]
This command is useful for single stepping through a section of code. The number of statements to single step could be specified. The user can walk single step only within the same procedure (WALK IN) or single step even in the called procedure (WALK 1). The default parameter is one so that the program stops after every line is executed. For example, suppose a user walks on the source code that looks as follows:

```plaintext
line#10: count = 10;
line#11: getvalue();
line#12: printf(" result= %d \n", count);
```

At line#10: WALK, WALK IN and WALK 1 are equivalent. Variable count is set to 10 and execution stopped at line #11.

At line#11: WALK IN will execute the getvalue procedure and stop at line #12. WALK will stop at the first line in the getvalue procedure.

At line#12: WALK has no meaning in the non-systems programming environment. Fdb will not single step the printf routine, and WALK IN and WALK are equivalent.

QUIT

The QUIT command causes you to exit the fdb.

TRACE

In BNF notation, trace is defined as:

```
<trace> ::= TRACE EXECUTION ;
```
This command is used to display the code-segment labels (code statement line numbers) encountered during program execution. This will also display the source lines. For example:

TRACE EXECUTION :print every line of code executed

RESTART

In BNF notation, restart is defined as:

```plaintext
<restart> ::= RESTART <option><parameter><file name> ;
<option> ::= <empty> | -<option char> ;
<option char> ::= <alpha> ;
<parameter> ::= <identifier> ;
```

This command is used to restart the debugged program. The user can set up options and parameters for the debugged program and also redirect the standard input/output device for the debugged program.

Suppose a user wants to debug a load module called compiler, whose option is -o and its parameter (file name to save the objects) is compile.o. Type this:

```plaintext
fdb compiler
RESTART -o compile.o
```

There are two types of output during a debug session. One is diagnostic messages from fdb and the other is output from the debugged program.

Fdb allows you to redirect either output. FILE command is used to redirect the debug messages and RESTART is used to redirect the program output.
If a user just presses the RETURN key (Newline Command), it is interpreted as if the previous command was entered.

Because of the newline feature and the multiple commands line feature, a command line that ends with a semicolon is different than one that ends without it. For example:

command 1       :this is just one command
command 1;      :this is equivalent to command 1 ; command 1
W               :single step execution command
(RETURN)        :execute next statement
(RETURN)        :execute next statement

SPECIAL CHARACTERS IN A STRING

A quote in a string is represented by two quotes. So "abc""d" is a string of abc"d", and "" is "". But "" is an illegal string.

A backslash (\) is used to indicate that a special character is following. So \ means single \. It is advised to use a backslash whenever non-alphanumeric characters are used. This does not apply in ALIAS replacement string.

If \ precedes %, EQU expansion is suppressed. For example:

EQU A "XYZ"
FIND "%A"    :search for XYZ
FIND " \%A"  :search for %A

The following example could cause a permanent loop, but will be detected and reported by fdb.

EQU a " \%a"  :define itself
%a            :would-be permanent expansion
Operating Procedure

The steps of a general operating procedure is described here. First the syntax of fdb is reviewed.

The syntax for calling fdb is:

\[ \text{fdb } [\text{object-file}[\text{directory}]] \]

where:

object-file: an executable program file which has been compiled with the -g (debug) option. The default for object-file is a.out.

directory: a directory where the source file exists. The default for directory is the working directory.

At any time there is a current line and current file. The current file may be changed by FILE command.

These are the steps in the procedure:

1. Compile source programs with -g option
2. Run loader
3. Run fdb

Suppose a C program is saved in test.c and a PASCAL program is in sample.p, and you try to debug the linked program (UNIX command syntax may be changed from time to time). These are the steps you follow.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -g test.c -o object</td>
<td>/* compile test.c program */</td>
</tr>
<tr>
<td>pc -g sample.p -o pobject</td>
<td>/* compile sample.p program */</td>
</tr>
</tbody>
</table>
If fdb has a bug and causes a permanent loop, you can't get out from fdb by pushing the Cancel key. In this case, hold down the Cancel key about 10 seconds. Then you can get out from fdb and return to the Unix shell.
The Fortune Operating System provides a number of utilities and libraries which make routine programming activities easier and less time consuming. In this section you will learn about the utilities and libraries below.

- Archive -ar
- Link Editor -ld
- Make
- Name -nm
- Ranlib
- Size
- Strip
- Libraries
  - libc.a
  - libg.a
  - libm.a

You will also learn about aspects of using C on the Fortune 32:16 which are specific to a 68000 based product.
Archive ar

Ar is used primarily to create and update library files used by the loader. Groups of files are maintained in one archive file. This version of ar uses an ASCII-format archive which can be ported among various machines running UNIX.

SYNTAX: ar key posname afile names(s)...

Element Purpose

key One character from the set of options (d, r, q, t, p, m, x). It can be catenated and enhanced with one or more of another set of options (v, u, a, i, b, c, l).

posname The filename you use to indicate position.

afile The name for the archive file.

name(s) The files in the archive file.

Each of the key options is described below.

Option Description

d Deletes the named files from the archive file.

r Replaces the named files in the archive file. If you include the optional character u only those files modified later than the archive files are replaced. If you use an optional positioning character from the set abi, then the posname argument must be included.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>Quickly appends the named files to the end of the archive file, disregarding any optional positioning characters and without checking whether the added files are already in the archive. When you are creating a large archive in pieces, use this to avoid quadratic behavior.</td>
</tr>
<tr>
<td>t</td>
<td>Prints a table of contents of the archive file. If no names are printed, all the files in the archive are tabled. If names are printed, only those files are tabled.</td>
</tr>
<tr>
<td>p</td>
<td>Prints the named files in the archive.</td>
</tr>
<tr>
<td>m</td>
<td>Moves the named files to the end of the archive. If you include a positioning character, then the posname argument must be present and, as with r, must specify where the files are to be moved.</td>
</tr>
<tr>
<td>x</td>
<td>Extracts the named files. If you give no names, all files in the archive are extracted. x does not, however, alter the archive file.</td>
</tr>
<tr>
<td>v</td>
<td>With the verbose option, you receive a file-by-file description of the construction of a new archive file. If you include t a listing of all information about the files will be included. With p each file is preceded by a name.</td>
</tr>
<tr>
<td>c</td>
<td>The create option suppresses the usual message produced when afile is created.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>1</td>
<td>The local option places files in the local directory rather than in /tmp, where it normally places temporary files.</td>
</tr>
</tbody>
</table>
The link editor, or loader, combines several object programs into one, resolves external references, and searches libraries. In the simplest form several object files are given and `ld` combines them. An object module is produced. It can be executed or used with the `-r` option as input for a further `ld` run. Output of `ld` is left in the `a.out` file (unless the `-o` option is used to specify an output filename) and is executable only if no errors occurred during loading.

SYNTAX: `ld` option files...

Argument routines are concatenated in the order you specify. Unless you use the `-e` option the entry point of the input of the executable or `a.out` file is the beginning of the first argument.

If any argument is a library, it is searched only once when it is encountered in the argument list. Only routines that define unresolved external references are loaded. The order of programs within libraries may be important. For example, if a routine from a library references another routine in the library, and the library has not been processed by `ranlib(1)`, the referenced routine must appear after the referencing routine in the library. The first member of a library should be a file named `__SYMDEF`. It is understood to be a dictionary for the library as produced by `ranlib(1)` and is searched iteratively to satisfy as many references as possible.

The symbols `etext`, `edata` and `end` are reserved, and if referenced, are set to the first location above the program, the first location above initialized data, and the first location above all data respectively. Don't define these symbols.
<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option</td>
<td>\textit{ld} understands several options (D, d, e, lx, M, N, n, o, r, s) and except for -l (this is the letter l), they should appear before the file names.</td>
</tr>
<tr>
<td>Files</td>
<td>These files are to be combined into the object module.</td>
</tr>
</tbody>
</table>

The following is a description of the link editor options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-D</td>
<td>Takes the next argument as a hexadecimal number and pads the data segment with zero bytes to the length you indicate.</td>
</tr>
<tr>
<td>-d</td>
<td>Forces definition of common storage even if the -r flag is included.</td>
</tr>
<tr>
<td>-e</td>
<td>The following argument becomes the entry point of the loaded program. Zero is the default. For example, with a program consisting of main( ) and main2( ):</td>
</tr>
</tbody>
</table>

You enter \[ \texttt{ld -e main2 filenames.o} \]

Result \[ \texttt{When you type a.out the program begins execution at main2.} \]

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-lx</td>
<td>This option is an abbreviation for the library name /lib/libx.a, where \textit{x} is a string. If that library doesn't exist, \texttt{ld} tries /usr/lib/libx.a. The library</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-M</td>
<td>Produces a primitive load map which lists the names of the files that will be loaded.</td>
</tr>
<tr>
<td>-N</td>
<td>The text portion is not made read-only or sharable. Uses &quot;magic number&quot; 0407.</td>
</tr>
<tr>
<td>-n</td>
<td>When the output file is executed, the text portion is made read-only. Therefore, it doesn't have to be repeated in memory if more than one copy of the program is being run concurrently. For example, if two or more people are expected to run an editor, loading it with -n can save space.</td>
</tr>
<tr>
<td>-o</td>
<td>Gives a name in the place of a.out to the ld output file.</td>
</tr>
<tr>
<td>-r</td>
<td>Relocation information is retained. This is useful for running the output through the loader again, if, for example, you don't include all files on the first run.</td>
</tr>
</tbody>
</table>

name must be placed last as it is searched for all undefined references when it is encountered.

You Enter | Results
---|---
ld filenames.o -lm | The math (m) library is searched.
You Enter
ld -r x.o y.o -o q.o
ld q.o z.o

Results
Puts results in q.o. The files x.o and y.o are combined with z.o to make a.out. This is the same as doing ld x.o y.o z.o

Option            Description

-S                Strips the output by removing all symbols but locals and globals.
-s                This is useful if you do not plan to reload, but only to execute. All symbol table and relocation information is removed, thereby saving space.
-T                The text segment origin is set by the next argument, a hexadecimal number.
-t                Traces the name of each file as it is processed and prints it on the screen.
-u                Takes the argument following and undefines it to force loading. This is useful for loading solely from a library.

You Enter
ld -u asin filenames.o

Results
asin would be included from the library you name.

Option            Description

-X                This discards any symbols that are not local, those whose names begin with ".".
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>Removes all local symbols and saves space in the output file.</td>
</tr>
</tbody>
</table>

You Enter

```
ld -x test.c
```

Results
test.o file that is smaller.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ysym</td>
<td>Lists each file in which sym appears, its type, and whether the file references or defines it.</td>
</tr>
</tbody>
</table>
When you are working on a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some of the source code. *Make* provides a simple means for maintaining up-to-date versions of programs. You can tell *make* a sequence of commands that creates certain files, and the list of files requiring other files to be current before the operations can be done. Whenever you make a change in any part of the program, use the *make* command to create the proper files simply, correctly, and with little effort.

Basically, *make* finds the name of a needed target in the description and ensures that all of the files that the target depends on are current. After ensuring that the supporting files are current, the target is made according to predefined instructions. If supporting files are not current, *make* will attempt to target each one. The description file defines the graph of dependencies. *Make* does a depth-first search of this graph and determines what work is really necessary.

In addition, *make* provides a simple macro substitution facility and the ability to condense commands into a single file for convenience. The *make* command takes four kinds of arguments: macro definitions, options, description, file names, and target file names.

**SYNTAX:** `make (options) (macro definitions) filenames...`

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>The options, from the set (i, s, r, n, t, q, p, d, f), are examined second, after the macro definition arguments.</td>
</tr>
</tbody>
</table>
A macro definition is a line including an equal sign not preceded by a colon or a tab. The name on the left of the equal sign (trailing blanks and tabs are stripped) is assigned to the string of characters to the right of the equal sign (tabs and leading blanks are stripped.) The following are examples:

```
CFLAGS = -I/u/james/mylib
```

The null string is a valid assignment.

Remaining arguments are the names of the targets to be made. They are processed left to right. If no such arguments exist, the first filename in the list of description files that doesn't begin with a period is made.

Following is a description of the options used with `make`.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i</td>
<td>Ignores error codes returned by invoked commands if the fake target name &quot;IGNORE&quot; is encountered in the description file.</td>
</tr>
<tr>
<td>-s</td>
<td>The silent mode doesn't print command lines before executing them. The same action is taken if the fake target name &quot;SILENT&quot; appears in the description file.</td>
</tr>
<tr>
<td>-r</td>
<td>Doesn't use the built-in rules.</td>
</tr>
<tr>
<td>-n</td>
<td>Commands are printed but not executed. Lines beginning with &quot;@&quot; are also printed.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-t</td>
<td>Updates (touches) the target files rather than issuing the normal commands.</td>
</tr>
<tr>
<td>-q</td>
<td>Questions whether the target file is or isn't up to date. Make returns a zero or non-zero status indicating up to date or not up to date.</td>
</tr>
<tr>
<td>-p</td>
<td>Prints the complete set of macro definitions and target descriptions.</td>
</tr>
<tr>
<td>-d</td>
<td>In debug mode make prints detailed information on files and times examined.</td>
</tr>
<tr>
<td>-f</td>
<td>The argument following f names a description file. The name &quot;-&quot; signifies standard input. If you include no -f arguments the file named makefile or Makefile in the current directory is read. When description files are present the contents override any built-in rules.</td>
</tr>
<tr>
<td>.DEFAULT</td>
<td>If a file must be made and no explicit commands or appropriate built-in rules exist, the commands in .DEFAULT are used if it exists.</td>
</tr>
<tr>
<td>.PRECIOUS</td>
<td>Doesn't remove dependents of this file if quit or interrupt is hit.</td>
</tr>
<tr>
<td>.SILENT</td>
<td>This has the same effect as the -s option.</td>
</tr>
<tr>
<td>.IGNORE</td>
<td>This has the same effect as the -i option.</td>
</tr>
</tbody>
</table>
Name nm

Name prints the symbol table of each object file in the list of arguments. A listing for each object file in the archive is produced if an argument is an archive.

Each symbol name is preceded by its value (blanks if undefined) and one of the following letters: U (undefined), A (absolute), T (text segment symbol), D (data segment symbol), B (bss segment symbol), C (common symbol), f file name, or - for sdb symbol table entries. For local symbols (non-external) the type letter is in lowercase. Output is sorted alphabetically.

SYNTAX: nm -option file...

You may use several options with the name utility.

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>The set of options is (a, g, n, o, p, r, u).</td>
</tr>
<tr>
<td>Files</td>
<td>These files are the object of the command. The symbols in a.out are listed if no file is given.</td>
</tr>
</tbody>
</table>

The options control the listings. Each option is described below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>All symbols are included for printing.</td>
</tr>
<tr>
<td>-g</td>
<td>Prints only global symbols, not local or fdb.</td>
</tr>
<tr>
<td>-n</td>
<td>Sorts numerically rather than alphabetically.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-o</td>
<td>The file or archive element name precedes each output line rather than only the first.</td>
</tr>
<tr>
<td>-p</td>
<td>Prints in symbol-table order rather than sorting.</td>
</tr>
<tr>
<td>-r</td>
<td>Sorts in reverse order.</td>
</tr>
<tr>
<td>-u</td>
<td>Prints only undefined symbols.</td>
</tr>
</tbody>
</table>
Ranlib

Ranlib adds a table of contents named __SYMDEF to the beginning of the archive. This way the archive can be loaded more rapidly. ar(1) is used to reconstruct the archive, so that enough temporary file space is available in the file system containing the current directory.

SYNTAX: ranlib archive...

The ranlib utility uses archive files.

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Archive</td>
<td>This is the name of the archive file containing a collection of .o files.</td>
</tr>
</tbody>
</table>
Size

Size prints the decimal number of bytes required by the text, data, and bss portions, and the sum in hex and decimal, of each object-file argument.

SYNTAX: size object ...

The size utility uses the object file that you are measuring.

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>The name of the file you are measuring. If you do not specify a file, a.out is used.</td>
</tr>
</tbody>
</table>

To see the size of a particular program, enter the following:

<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>size test.c</td>
<td>text  data  bss  dec  hex</td>
</tr>
<tr>
<td></td>
<td>60    16     0    76    4c</td>
</tr>
</tbody>
</table>

You can do a comparison on file size by running size on a program before and after, using the optimizer which reduces code size.
Strip

Strip removes the symbol table and relocation bits which are usually attached to the output of the assembler and loader. Use this to save space after you have debugged a program. Strip has the same effect as the -s option of ld.

SYNTAX: strip name ...

The strip utility reduces the size of a file.

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The file you want to strip.</td>
</tr>
</tbody>
</table>
Tool Usage

The following procedure allows you to use these tools: size, nm, strip, ar, ranlib, make, and lint.

1 Create a C language program.

<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed x.c</td>
<td>A program named x.c is created which prints a message.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>a main ( ){</td>
<td></td>
</tr>
<tr>
<td></td>
<td>printf (&quot;Hello, World\n&quot;);</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td></td>
</tr>
<tr>
<td>cc x.c</td>
<td>Compiles the program.</td>
</tr>
<tr>
<td>ls -l a.out</td>
<td>Lists the output file.</td>
</tr>
<tr>
<td>a.out</td>
<td>Runs the program.</td>
</tr>
<tr>
<td>size a.out</td>
<td>Displays the size.</td>
</tr>
<tr>
<td>nm a.out</td>
<td>Prints the symbol table of the a.out object file.</td>
</tr>
<tr>
<td>strip a.out</td>
<td>Strips off the symbol table.</td>
</tr>
<tr>
<td>ls -l a.out</td>
<td>Shows that the program is smaller. (use size to show that the symbol table is gone.)</td>
</tr>
</tbody>
</table>

2 Now create two subroutines.

<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed hello.c</td>
<td>Creates a subroutine named hello.c.</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

66
hello ( ){
    printf ("Hello, World \n")
}
wq
ed goodbye.c Creates a subroutine named goodbye.c.
a
goodbye ( ){
    printf ("Goodbye, World \n");
}
.
wq

3 Compile the subroutines. Then create the main program that calls the subroutines.

You Enter Results

cc -c hello.c goodbye.c Compiles subroutines. Lists all
ls *.o .o files.
ed main.c
a
main (* ){
    hello ( );
}
.
wq
4 Compile the main program.

<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -c main.c</td>
<td>Compile the program.</td>
</tr>
<tr>
<td>ar crv greet.a hello.o goodbye.o</td>
<td>Create the archive even if it already exists.</td>
</tr>
<tr>
<td>ar tv greet.a</td>
<td>Prints table of contents of the library</td>
</tr>
<tr>
<td>ranlib greet.a</td>
<td>Inserts table of contents in front of library.</td>
</tr>
<tr>
<td>cc main.o greet.a</td>
<td>This link edits the archive of .o files with the main program.</td>
</tr>
<tr>
<td>nm a.out</td>
<td>Notice hello is in the name list and goodbye isn't.</td>
</tr>
</tbody>
</table>

5 Create a makefile. Use the make utility to create the a.out file.

<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed makefile</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>hello.o: hello.c</td>
<td></td>
</tr>
<tr>
<td>goodbye.o: goodbye.c</td>
<td></td>
</tr>
<tr>
<td>main.o: main.c</td>
<td></td>
</tr>
<tr>
<td>greet.a: hello.o goodbye.o</td>
<td></td>
</tr>
<tr>
<td>ar crv greet.a hello.o goodbye.o</td>
<td></td>
</tr>
<tr>
<td>main: main.o greet.a</td>
<td></td>
</tr>
<tr>
<td>cc -o main main.o greet.a</td>
<td></td>
</tr>
</tbody>
</table>

68
<table>
<thead>
<tr>
<th>You Enter</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td></td>
</tr>
<tr>
<td>make main</td>
<td></td>
</tr>
</tbody>
</table>

Compiles source files that have been changed.
Information on how to use the library functions, arguments, and returns can be found in Section 2. The Fortune Operating System contains numerous libraries designed for many different applications. Some are specifically for use with C. C related libraries are summarized below.

- libc.a is the general C library containing string, input/output, and system functions.
- libg.a contains support routines for the Fortune Symbolic Debugger (FDB).
- libm.a contains math, transcendental, and power functions.

To avoid conflict with library global names, do not use any of the following names for global variables or procedures.

<table>
<thead>
<tr>
<th>_dbargs</th>
<th>_filbuf</th>
<th>BC</th>
<th>asctime</th>
</tr>
</thead>
<tbody>
<tr>
<td>_dbsubc</td>
<td>_flsbuf</td>
<td>PC</td>
<td>asin</td>
</tr>
<tr>
<td>_dbsubn</td>
<td>_getccl</td>
<td>UP</td>
<td>atan</td>
</tr>
<tr>
<td>_callc</td>
<td>_innum</td>
<td>abort</td>
<td>atan2</td>
</tr>
<tr>
<td>_calle</td>
<td>_instr</td>
<td>abs</td>
<td>atoi</td>
</tr>
<tr>
<td>_error</td>
<td>_iob</td>
<td>access</td>
<td>atol</td>
</tr>
<tr>
<td>_cleanup</td>
<td>_lastbuf</td>
<td>acct</td>
<td>atof</td>
</tr>
<tr>
<td>_cret</td>
<td>_regbak</td>
<td>acos</td>
<td>auldiv</td>
</tr>
<tr>
<td>_csav</td>
<td>_regsav</td>
<td>alarm</td>
<td>aulmul</td>
</tr>
<tr>
<td>_csavl</td>
<td>_sctab</td>
<td>aldiv</td>
<td>aulrem</td>
</tr>
<tr>
<td><em>ctype</em></td>
<td>_sibuf</td>
<td>allocp</td>
<td>blt</td>
</tr>
<tr>
<td>_doprnt</td>
<td>_sighnd</td>
<td>allocs</td>
<td>brk</td>
</tr>
<tr>
<td>_doscan</td>
<td>_sobuf</td>
<td>alloct</td>
<td>cabs</td>
</tr>
<tr>
<td>_e10tab</td>
<td>_strout</td>
<td>allocx</td>
<td>callloc</td>
</tr>
<tr>
<td>_error</td>
<td>l3tol</td>
<td>almul</td>
<td>ceil</td>
</tr>
<tr>
<td>_exit</td>
<td>ltol3</td>
<td>alrem</td>
<td>cfree</td>
</tr>
</tbody>
</table>
Fortune 32:16 Specific Aspects

There are four machine-specific qualities of the Fortune C compiler. Each is explained below.

INTEGERS AND POINTERS

Types Integer, pointer, and long are each 32 bits long. The type short is 16 bits long. Character data is 8 bits long. Unsigned data is the same length as the corresponding signed quantities.

SIGN EXTENSION

Character data is sign extended unless the user declares unsigned character.

BYTE ORDERING

The Motorola 68000 addresses bytes sequentially from high to low order. If you reference the address pointer of an integer (int) as a character (char) you will get the high order byte of the integer (the most significant portion).

ALIGNMENT

All variables and structures are aligned to even byte addresses and occupy an even number of bytes. To maintain machine independence when coding in C, be aware of the following issues.

- The length of int may not be the same as anything else, such as a pointer, a long, or a short.

- Addressing should not be done within a basic type.
• Calculating addresses should not be done within a structure.

• The type char may not be sign extended in all calculations.

• Nothing should be accessed within the local frame area, except with declared names.
The optimizer can increase the throughput of your programs. To get the best out of your Fortune optimizer follow these rules.

1 Use register variables as much as possible, especially floating point, to affect code size and speed.

2 Use shorts whenever possible. Although the compiler may occasionally have to extend them, operations with shorts are much faster than character or integer equivalents (except byte moves).

3 Use of logical operations, such as $x$ and $y$ where $y$ is a constant, optimize better than subtraction or comparison. The same is true for the operator.

4 Structures or array references, especially byte arrays, are optimized if their lengths are powers of two.

5 The C language has no common subexpression or invariant code optimizer. Place only necessary expressions inside loops and do not repeat expressions in straight line code.

6 Use register variables in a function only if the variable is used in a loop or is used at least four times in the function for the first register, and three times for succeeding registers.

7 Register variables should be kept on the left-hand side of the expression. For example, write

$$ r = f + g; \quad (r \text{ and } f \text{ are register vars}) $$
rather than

\[ r = g + f \]

8 Generally, automatics access more quickly than static variables. However, heavily used statics may produce better code than automatic variables.

- If the variable or array is referenced more than three times, place it in static (unless there are no other register variables).

- If the variable is a structure avoid placing it in static.

9 If your program will not allocate more than 8K of stack space, you may compile with the -G option which reduces stack growth and checks calls at every procedure invocation.

10 Use short multiplication and division whenever possible. Cast or convert everything to shorts before doing the operation to ensure the use of hardware instructions. Multiplication or division by a power of two is converted to shifts, however.

11 When moving approximately one half or more of a structure use a full structure move, and then restore the contents. Use full structure move whenever possible.

12 Keep for loops simple, using one variable going through a simple range, rather than lots of conditionals. Use a simple increment such as ++g.
The C language is well suited for high-speed system applications. Fortran is designed for mathematical and scientific applications. You may find it desirable to write multi-language applications that use the strengths of each language. You may write a program in Fortran, for example, that calls a graphics package written in C. With the language interface capabilities on the Fortune 32:16, C procedures can be written to call or to be called by Fortran procedures. To do this you must know the rules that completed code obeys for procedure names, data representation, return values, and argument lists.

Procedure Names

On UNIX systems the name of a Fortran procedure or a common block is represented as seen. It is accessible from other languages without any additional notation.

Data Representation

The following table shows corresponding Fortran and C declarations.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer*2 x</td>
<td>short int x;</td>
</tr>
<tr>
<td>integer x</td>
<td>long int x;</td>
</tr>
</tbody>
</table>
In Fortran, integer, logical, and real data occupy the same amount of memory.

Return Values

A function in Fortran of type integer, logical, real, or double precision will return the same value as a C routine of the corresponding type. A complex or double complex function in Fortran is equivalent to a C routine that includes an additional argument pointing to the location where the return value is to be stored. In this example,

```fortran
complex function sin(. . . )
```

is equivalent to

```fortran
sin (temp, . . . )
struct float r, i; *temp;
```

A character-valued function in Fortran is the same as a C routine which includes two extra initial arguments: a data address and a length. For example,

```fortran
character *15 function strcpy(. . . )
```

is the same as
strcpy(result, length, ...)
char result  ;
long int length;

and could be called in C with

char chars  15 ;
   ...
strcpy(chars, 15L, ...);

Subroutines are called as if they were integer-valued functions whose value indicates which alternate return value to use. The alternate return arguments are labels and are not passed to the function. They are used to do an indexed branch in the calling procedure. If the subroutine provides no entry points with alternate return arguments, the returned value is not defined.

In this example, the statement

call nref(*10, *20, *30)

is treated as if were

goto (10, 20, 30), nret()

Arguments Lists

Fortran arguments are passed by address. Also, all type char and dummy procedure arguments pass an argument giving the length of the value. String lengths are long int quantities passed by value. Arguments are given in the following order:

- Additional arguments for complex and character functions
- Address for each item of data or function
- A long integer for each character or procedure argument
The call in

    external f
    character*7 s
    integer b(3)
    ...
    call sam(f, b(2), s)

is the same as

    int f();
    char s[7];
    long int b 3 ;
    ...
    sam (f, &b 1 , s, 0L, 7L);

The first element of a C array has the subscript zero, whereas Fortran arrays begin at one. Also, Fortran arrays are stored in column-major order; C arrays are stored in row-major order.
Part 1  System Routines

This set of routines provides the interface of the C language to the UNIX operating system. Using these routines you will be able to access many of the UNIX system calls by way of C programs.
NAME
intro, errno - introduction to system calls and error numbers

SYNOPSIS
#include <errno.h>

DESCRIPTION
Section 2 of this manual describes all the entries into the system. Distinctions as to the status of the entries are made in the headings:

(2) System call entries which are standard in Version 7 UNIX systems.

(2J) System call entries added in support of the job control mechanisms of csh(1). These system calls are not available in standard Version 7 UNIX systems, and should be used only when necessary; to prevent implicit use they are contained in the jobs library which must be specifically requested with the -ljobs loader option. The use of conditional compilation is recommended when possible so that programs which use these features will gracefully degrade on systems which lack job control.

(2V) System calls added for the Virtual Memory version of UNIX distributed by Berkeley. Some of these calls are likely to be replaced by new facilities in future versions; in cases where this is imminent, this is indicated in the individual manual pages.

An error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections specify the details. An error number is also made available in the external variable errno. Errno is not cleared on successful calls, so it should be tested only after an error has occurred.

There is a table of messages associated with each error, and a routine for printing the message; See perror(3). The possible error numbers are not recited with each writeup in section 2, since many errors are possible for most of the calls. Here is a list of the error numbers, their names as defined in <errno.h>, and the messages available using perror.

0       Error 0
    Unused.
1 EPERM  Not owner
   Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT  No such file or directory
   This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH  No such process
   The process whose number was given to signal and ptrace does not exist, or is already dead.

4 EINTR  Interrupted system call
   An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO  I/O error
   Some physical I/O error occurred during a read or write. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO  No such device or address
   I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialed in or no disk pack is loaded on a drive.

7 E2BIG  Arg list too long
   An argument list longer than 10240 bytes is presented to exec.

8 ENOEXEC  Exec format error
   A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see a.out(5).

9 EBADF  Bad file number
   Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which is open only for writing (resp. reading).

10 ECHILD  No children
   Wait and the process has no living or unwaited-for children.
11 **EAGAIN** No more processes
   In a fork, the system's process table is full or the
   user is not allowed to create any more processes.

12 **ENOMEM** Not enough core
   During an exec or break, a program asks for more core
   than the system is able to supply. This is not a tem-
   porary condition; the maximum core size is a system
   parameter. The error may also occur if the arrangement
   of text, data, and stack segments requires too many
   segmentation registers.

13 **EACCES** Permission denied
   An attempt was made to access a file in a way forbidden
   by the protection system.

14 **EFAULT** Bad address
   The system encountered a hardware fault in attempting
   to access the arguments of a system call.

15 **ENOTBLK** Block device required
   A plain file was mentioned where a block device was
   required, e.g. in mount.

16 **EBUSY** Mount device busy
   An attempt to mount a device that was already mounted
   or an attempt was made to dismount a device on which
   there is an active file directory. (open file, current
   directory, mounted-on file, active text segment).

17 **EXIST** File exists
   An existing file was mentioned in an inappropriate con-
   text, e.g. link.

18 **EXDEV** Cross-device link
   A link to a file on another device was attempted.

19 **ENODEV** No such device
   An attempt was made to apply an inappropriate system
   call to a device; e.g. read a write-only device.

20 **ENOTDIR** Not a directory
   A non-directory was specified where a directory is
   required, for example in a path name or as an argument
   to chdir.

21 **EISDIR** Is a directory
   An attempt to write on a directory.

22 **EINVAL** Invalid argument
   Some invalid argument: dismounting a non-mounted
device, mentioning an unknown signal in signal, reading or writing a file for which seek has generated a negative pointer. Also set by math functions, see intro(3).

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more opens can be accepted.

24 EMFILE Too many open files
Customary configuration limit is 20 per process.

25 ENOTTY Not a typewriter
The file mentioned in stty or gtty is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum (about 1.0E9 bytes).

28 ENOSPC No space left on device
During a write to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An lseek was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 32767 links to a file.

32 EPipe Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.
34  ERANGE  Result too large
    The value of a function in the math package (3M) is
    unrepresentable within machine precision.

SEE ALSO
intro(3)

BUGS
    The message "Mount device busy" is reported when a terminal
    is inaccessible because the "exclusive use" bit is set; this
    is confusing.
NAME
access - determine accessibility of file

SYNOPSIS
access(name, mode)
char *name;

DESCRIPTION
Access checks the given file name for accessibility according to mode, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. Specifying mode 0 tests whether the directories leading to the file can be searched and the file exists.

An appropriate error indication is returned if name cannot be found or if any of the desired access modes would not be granted. On disallowed accesses -1 is returned and the error code is in errno. 0 is returned from successful tests.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by access, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but exec will fail unless it is in proper format.

SEE ALSO
stat(2)
NAME
acct - turn accounting on or off

SYNOPSIS
acct(file)
char *file;

DESCRIPTION
The system is prepared to write a record in an accounting file for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to file. An argument of 0 causes accounting to be turned off.

The accounting file format is given in acct(5).

SEE ALSO
acct(5), sa(8)

DIAGNOSTICS
On error -1 is returned. The file must exist and the call may be exercised only by the super-user. It is erroneous to try to turn on accounting when it is already on.

BUGS
No accounting is produced for programs running when a crash occurs. In particular nonterminating programs are never accounted for.
NAME
alarm - schedule signal after specified time

SYNOPSIS
alarm(seconds)
unsigned seconds;

DESCRIPTION
Alarm causes signal SIGALRM, see signal(2), to be sent to
the invoking process in a number of seconds given by the
argument. Unless caught or ignored, the signal terminates
the process.

Alarm requests are not stacked; successive calls reset the
alarm clock. If the argument is 0, any alarm request is
canceled. Because the clock has a 1-second resolution, the
signal may occur up to one second early; because of schedul­
ing delays, resumption of execution of when the signal is
called may be delayed an arbitrary amount. The longest
specifiable delay time is 2147483647 seconds.

The return value is the amount of time previously remaining
in the alarm clock.

SEE ALSO
pause(2), signal(2), sigsys(2), sigset(3), sleep(3)
NAME
brk, sbrk - change core allocation

SYNOPSIS
char *brk(addr)
char *sbrk(incr)

DESCRIPTION
Brk sets the system's idea of the lowest location not used by the program (called the break) to addr (rounded up to the next multiple of 1024 bytes). Locations not less than addr and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function sbrk, incr more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via exec the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use break.

The vlimit(2) system call may be used to determine the maximum permissible size of the data region; it will not be possible to set the break beyond "etext + vlimit(LIM_DATA, -1)." (See end(3) for the definition of etext.)

SEE ALSO
exec(2), vlimit(2), malloc(3), end(3)

DIAGNOSTICS
Zero is returned if the brk could be set; -1 if the program requests more memory than the system limit or if too many segmentation registers would be required to implement the break. Sbrk returns -1 if the break could not be set.
NAME
   chdir - change current working directory

SYNOPSIS
   chdir(dirname)
   char *dirname;

DESCRIPTION
   Dirname is the address of the pathname of a directory, termi­
   nated by a null byte.  Chdir causes this directory to
   become the current working directory, the starting point for
   path names not beginning with '/'.

SEE ALSO
   cd(1)

DIAGNOSTICS
   Zero is returned if the directory is changed; -1 is returned
   if the given name is not that of a directory or is not
   searchable.
NAME
chmod - change mode of file

SYNOPSIS
chmod(name, mode)
char *name;

DESCRIPTION
The file whose name is given as the null-terminated string
pointed to by name has its mode changed to mode. Modes are
constructed by oring together some combination of the fol­
lowing:

- 04000 set user ID on execution
- 02000 set group ID on execution
- 01000 save text image after execution
- 00400 read by owner
- 00200 write by owner
- 00100 execute (search on directory) by owner
- 00070 read, write, execute (search) by group
- 00007 read, write, execute (search) by others

If an executable file is set up for sharing (this is the
default) then mode 1000 prevents the system from abandoning
the swap-space image of the program-text portion of the file
when its last user terminates. Ability to set this bit is
restricted to the super-user since swap space is consumed by
the images. See sticky(8).

Only the owner of a file (or the super-user) may change the
mode. Only the super-user can set the 1000 mode.

On some systems, writing or changing the owner of a file
turns off the set-user-id bit. This makes the system some­
what more secure by protecting set-user-id files from
remaining set-user-id if they are modified, at the expense
of a degree of compatibility.

SEE ALSO
chmod(1)

DIAGNOSTIC
Zero is returned if the mode is changed; -1 is returned if
name cannot be found or if the current user is neither the
owner of the file nor the super-user.
NAME
   chown - change owner and group of a file

SYNOPSIS
   chown(name, owner, group)
   char *name;

DESCRIPTION
   The file whose name is given by the null-terminated string
   pointed to by name has its owner and group changed as speci­
   fied. Only the super-user may execute this call, because if
   users were able to give files away, they could defeat the
   (nonexistent) file-space accounting procedures.

   On some systems, chown clears the set-user-id bit on the
   file to prevent accidental creation of set-user-id programs
   owned by the super-user.

SEE ALSO
   chown(1), passwd(5)

DIAGNOSTICS
   Zero is returned if the owner is changed; -1 is returned on
   illegal owner changes.
NAME
    close - close a file

SYNOPSIS
    close(fildes)

DESCRIPTION
    Given a file descriptor such as returned from an open, creat, dup or pipe(2) call, close closes the associated file. A close of all files is automatic on exit, but since there is a limit on the number of open files per process, close is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain high-numbered file descriptors are closed by exec(2), and it is possible to arrange for others to be closed (see PIOCLEX in ioctl(2)).

SEE ALSO
    creat(2), open(2), pipe(2), exec(2), ioctl(2)

DIAGNOSTICS
    Zero is returned if a file is closed; -1 is returned for an unknown file descriptor.

BUGS
    A file cannot be closed while there are pages which have been vread but not referenced.
NAME
creat - create a new file

SYNOPSIS
creat(name, mode)
char *name;

DESCRIPTION
Creat creates a new file or prepares to rewrite an existing file called name, given as the address of a null-terminated string. If the file did not exist, it is given mode mode, as modified by the process's mode mask (see umask(2)). Also see chmod(2) for the construction of the mode argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned.

The mode given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a creat, an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO
write(2), close(2), chmod(2), umask (2)

DIAGNOSTICS
The value -1 is returned if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

BUGS
A file cannot be truncated while any process has pages set up by a vread on that file which have not been referenced.
NAME
dup, dup2 - duplicate an open file descriptor

SYNOPSIS
dup(fildes)
int fildes;

dup2(fildes, fildes2)
int fildes, fildes2;

DESCRIPTION
Given a file descriptor returned from an open, pipe, or creat call, dup allocates another file descriptor synonymous with the original. The new file descriptor is returned.

In the second form of the call, fildes is a file descriptor referring to an open file, and fildes2 is a non-negative integer less than the maximum value allowed for file descriptors (approximately 19). Dup2 causes fildes2 to refer to the same file as fildes. If fildes2 already referred to an open file, it is closed first.

SEE ALSO
creat(2), open(2), close(2), pipe(2)

DIAGNOSTICS
The value -1 is returned if: the given file descriptor is invalid; there are already too many open files.

BUGS
Dup2 fails if fildes2 was vread from and some of the pages have not been referenced.
NAME
execl, execv, execle, execve, execlp, execvp, exece, environ
- execute a file

SYNOPSIS
execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[];

execle(name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];

execve(name, argv, envp)
char *name, *argv[], *envp[];

extern char **environ;

DESCRIPTION
Exec in all its forms overlays the calling process with the
named file, then transfers to the entry point of the core
image of the file. There can be no return from a successful
exec; the calling core image is lost.

Files remain open across exec unless explicit arrangement
has been made; see ioctl(2). Ignored/held signals remain
ignored/held across these calls, but signals that are caught
(see signal(2)) are reset to their default values.

Each user has a real user ID and group ID and an effective
user ID and group ID. The real ID identifies the person
using the system; the effective ID determines his access
privileges. Exec changes the effective user and group ID to
the owner of the executed file if the file has the 'set-
user-ID' or 'set-group-ID' modes. The real user ID is not
affected.

The name argument is a pointer to the name of the file to be
executed. The pointers arg[0], arg[1] ... address null-
terminated strings. Conventionally arg[0] is the name of
the file.

From C, two interfaces are available. execl is useful when
a known file with known arguments is being called; the argu-
ments to execl are the character strings constituting the
file and the arguments; the first argument is conventionally
the same as the file name (or its last component). A 0
argument must end the argument list.
The `execv` version is useful when the number of arguments is unknown in advance; the arguments to `execv` are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a `0` pointer.

When a C program is executed, it is called as follows:

```c
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where `argc` is the argument count and `argv` is an array of character pointers to the arguments themselves. As indicated, `argc` is conventionally at least one and the first member of the array points to a string containing the name of the file.

`argv` is directly usable in another `execv` because `argv[argc]` is `0`.

`Envp` is a pointer to an array of strings that constitute the environment of the process. Each string consists of a name, an `=`", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell `sh(1)` passes an environment entry for each global shell variable defined when the program is called. See `environ(5)` for some conventionally used names. The C run-time start-off routine places a copy of `envp` in the global cell `environ`, which is used by `execv` and `exec1` to pass the environment to any sub-programs executed by the current program. The `exec` routines use lower-level routines as follows to pass an environment explicitly:

```c
execve(file, argv, environ);
execle(file, arg0, arg1, ... , argn, 0, environ);
```

`Execlp` and `execvp` are called with the same arguments as `execl` and `execv`, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

To aid execution of command files of various programs, if the first two characters of the executable file are `#!` then `exec` attempts to read a pathname from the executable file and use that program as the command files command interpreter. For example, the following command file sequence would be used to begin a `csh` script:

```c
#!/bin/csh
# This shell script computes the checksum on /dev/foobar
#
...```
A single parameter may be passed the interpreter, specified after the name of the interpreter; its length and the length of the name of the interpreter combined must not exceed 32 characters. The space (or tab) following the '#!' is mandatory, and the pathname must be explicit (no paths are searched).

FILES
/bin/sh  shell, invoked if command file found by execvp or execv

SEE ALSO
fork(2), environ(5), csh(1)

DIAGNOSTICS
If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see a.out(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is -1. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

BUGS
If execvp is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of argv[0] and argv[-1] will be modified before return.
NAME
 exit - terminate process

SYNOPSIS
 exit(status)
 int status;

_exit(status)
 int status;

DESCRIPTION
 Exit is the normal means of terminating a process. Exit closes all the process's files and notifies the parent process if it is executing a wait. The low-order 8 bits of status are available to the parent process.

This call can never return.

The C function exit may cause cleanup actions before the final 'sys exit'. The function _exit circumvents all cleanup, and should be used to terminate a child process after a fork(2) or vfork(2) to avoid flushing buffered output twice.

SEE ALSO
 fork(2), vfork(2), wait(2)
NAME
fork - spawn new process

SYNOPSIS
fork()

DESCRIPTION
Fork and vfork(2) are the only ways new processes are created. With fork, the new process's core image is a copy of that of the caller of fork. The only distinction is the fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. This process ID is used by wait(2).

Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

Vfork is the most efficient way of creating a new process when the fork is to be followed shortly by an exec, but is not suitable when the fork is not to be followed by an exec.

SEE ALSO
wait(2), exec(2), vfork(2)

DIAGNOSTICS
Returns -1 and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full. Only the super-user can take the last process-table slot.
NAME
  getpid - get process identification

SYNOPSIS
  getpid()

DESCRIPTION
  getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

SEE ALSO
  mktemp(3)
NAME
  getuid, getgid, geteuid, getegid - get user and group identity

SYNOPSIS
  getuid()
  geteuid()
  getgid()
  getegid()

DESCRIPTION
  Getuid returns the real user ID of the current process, geteuid the effective user ID. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the 'set user ID' mode, to find out who invoked them.

  Getgid returns the real group ID, getegid the effective group ID.

SEE ALSO
  setuid(2)
IOCTL(2)       System Routines       IOCTL(2)

NAME
ioctl, stty, gtty - control device

SYNOPSIS
#include <sgtty.h>

ioctl(fildes, request, argp)
struct sgttyb *argp;

stty(fildes, argp)
struct sgttyb *argp;

gtty(fildes, argp)
struct sgttyb *argp;

DESCRIPTION
ioctl performs a variety of functions on character special
files (devices). The writeups of various devices in section
4 discuss how ioctl applies to them.

For certain status setting and status inquiries about termi­
nal devices, the functions stty and gtty are equivalent to
ioctl(fildes, TIOCSETP, argp)
ioctl(fildes, TIOCGETP, argp)
respectively; see tty(4).

The following two standard calls, however, apply to any open
file:

ioctl(fildes, FIOCLEX, NULL);
ioctl(fildes, FIONCLEX, NULL);

The first causes the file to be closed automatically during
a successful exec operation; the second reverses the effect
of the first.

The following call is peculiar to the Berkeley implementa­
tion, and also applies to any open file:

ioctl(fildes, FIONREAD, &count)

returning, in the longword count the number of characters
available for reading from fildes.

SEE ALSO
stty(1), tty(4), exec(2)

DIAGNOSTICS
Zero is returned if the call was successful; -1 if the file
descriptor does not refer to the kind of file for which it
was intended, or if request attempts to modify the state of a terminal when fildes is not writeable.

**ioctl** calls which attempt to modify the state of a process control terminal while a process is not in the process group of the control terminal will cause a SIGTTOU signal to be sent to the process' process group. Such **ioctl**s are allowed, however, if SIGTTOU is being held, ignored, if the process is an orphan which has been inherited by **init**, or is the child in an incomplete **vfork** (see **jobs(3)**)

**BUGS**

Strictly speaking, since **ioctl** may be extended in different ways to devices with different properties, argp should have an open-ended declaration like

```
union { struct sgttyb ...; ... } *argp;
```

The important thing is that the size is fixed by `struct sgttyb`.
NAME
   kill - send signal to a process

SYNOPSIS
   kill(pid, sig)

DESCRIPTION
   Kill sends the signal sig to the process specified by the
   process number pid. See sigsys(2) for a list of signals.

   The sending and receiving processes must have the same
   effective user ID, otherwise this call is restricted to the
   super-user. (A single exception is the signal SIGCONT which
   may be sent as described in killpg(2), although it is usu­
   ally sent using killpg rather than kill).

   If the process number is 0, the signal is sent to all other
   processes in the sender's process group; see tty(4) and also
   killpg.

   If the process number is -1, and the user is the super-user,
   the signal is broadcast universally except to processes 0,
   1, 2, the scheduler initialization, and pageout processes,
   and the process sending the signal.

   Processes may send signals to themselves.

SEE ALSO
   sigsys(2), signal(2), kill(1), killpg(2), init(8)

DIAGNOSTICS
   Zero is returned if the process is killed; -1 is returned if
   the process does not have the same effective user ID and the
   user is not super-user, or if the process does not exist.
NAME
  link - link to a file

SYNOPSIS
  link(namel, name2)
  char *namel, *name2;

DESCRIPTION
  A link to namel is created; the link has the name name2.
  Either name may be an arbitrary path name.

SEE ALSO
  ln(1), unlink(2)

DIAGNOSTICS
  Zero is returned when a link is made; -1 is returned when
  namel cannot be found; when name2 already exists; when the
  directory of name2 cannot be written; when an attempt is
  made to link to a directory by a user other than the super-
  user; when an attempt is made to link to a file on another
  file system; when a file has too many links.

  On some systems the super-user may link to non-ordinary
  files.
NAME
lseek, tell - move read/write pointer

SYNOPSIS
long lseek(int fildes, off_t offset, int whence)
long offset;

long tell(int fildes)

DESCRIPTION
The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

If whence is 0, the pointer is set to offset bytes.

If whence is 1, the pointer is set to its current location plus offset.

If whence is 2, the pointer is set to the size of the file plus offset.

The returned value is the resulting pointer location.

The obsolete function tell(fildes) is identical to lseek(fildes, 0L, 1).

Seeking far beyond the end of a file, then writing, creates a gap or 'hole', which occupies no physical space and reads as zeros.

SEE ALSO
open(2), creat(2), fseek(3)

DIAGNOSTICS
-1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file.

BUGS
lseek is a no-op on character special files.
NAME
mknod – make a directory or a special file

SYNOPSIS
mknod(name, mode, addr)
char *name;

DESCRIPTION
Mknod creates a new file whose name is the null-terminated string pointed to by name. The mode of the new file (including directory and special file bits) is initialized from mode. (The protection part of the mode is modified by the process's mode mask; see umask(2)). The first block pointer of the i-node is initialized from addr. For ordinary files and directories addr is normally zero. In the case of a special file, addr specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO
mkdir(1), mknod(1), filsys(5)

DIAGNOSTICS
Zero is returned if the file has been made; -1 if the file already exists or if the user is not the super-user.
NAME
mount, umount - mount or remove file system

SYNOPSIS
mount(special, name, rwflag)
char *special, *name;

umount(special)
char *special;

DESCRIPTION
Mount announces to the system that a removable file system
has been mounted on the block-structured special file special; from now on, references to file name will refer to the
root file on the newly mounted file system. Special and
name are pointers to null-terminated strings containing the
appropriate path names.

Name must exist already. Name must be a directory (unless
the root of the mounted file system is not a directory).
Its old contents are inaccessible while the file system is
mounted.

The rwflag argument determines whether the file system can
be written on; if it is 0 writing is allowed, if non-zero no
writing is done. Physically write-protected and magnetic
tape file systems must be mounted read-only or errors will
occur when access times are updated, whether or not any
explicit write is attempted.

Umount announces to the system that the special file is no
longer to contain a removable file system. The associated
file reverts to its ordinary interpretation.

SEE ALSO
mount(8)

DIAGNOSTICS
Mount returns 0 if the action occurred; -1 if special is
inaccessible or not an appropriate file; if name does not
exist; if special is already mounted; if name is in use; or
if there are already too many file systems mounted.

Umount returns 0 if the action occurred; -1 if the spe-
cial file is inaccessible or does not have a mounted file
system, or if there are active files in the mounted file
system.

BUGS
If a file containing holes (unallocated blocks) is read,
even on a file system mounted read-only, the system will
attempt to fill in the holes by writing on the device.
NAME
nice - set program priority

SYNOPSIS
nice incr

DESCRIPTION
The scheduling priority of the process is augmented by incr. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flak from the administration.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range -20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by fork(2). For a privileged process to return to normal priority from an unknown state, nice should be called successively with arguments -40 (goes to priority -20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SEE ALSO
nice(1), fork(2), renice(8)
NAME
open - open for reading or writing

SYNOPSIS
open(name, mode)
char *name;

DESCRIPTION
Open opens the file name for reading (if mode is 0), writing (if mode is 1) or for both reading and writing (if mode is 2). Name is the address of a string of ASCII characters representing a path name, terminated by a null character.

The file is positioned at the beginning (byte 0). The returned file descriptor must be used for subsequent calls for other input-output functions on the file.

SEE ALSO
creat(2), read(2), write(2), dup(2), close(2)

DIAGNOSTICS
The value -1 is returned if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open.

BUGS
It should be possible to optionally open files for writing with exclusive use, and to optionally call open without the possibility of hanging waiting for carrier on communication lines.
NAME
    pause - stop until signal

SYNOPSIS
    pause()

DESCRIPTION
    Pause never returns normally. It is used to give up control
    while waiting for a signal from kill(2) or alarm(2). Upon
    termination of a signal handler started during a pause, the
    pause call will return.

SEE ALSO
    kill(1), kill(2), alarm(2), sigsys(2), signal(2), sigset(3),
    setjmp(3)
NAME
pipe - create an interprocess channel

SYNOPSIS
pipe(fildes)
int fildes[2];

DESCRIPTION
The pipe system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor fildes[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor fildes[0] will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent fork calls) will pass data through the pipe with read and write calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

SEE ALSO
sh(1), read(2), write(2), fork(2)

DIAGNOSTICS
The function value zero is returned if the pipe was created; -1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

BUGS
Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.
NAME
ptrace - process trace

SYNOPSIS
#include <signal.h>

ptrace(request, pid, addr, data)
int *addr;

DESCRIPTION
Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a request argument. Generally, pid is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt.' See signal(2) for the list. Then the traced process enters a stopped state and its parent is notified via wait(2). When the child is in the stopped state, its core image can be examined and modified using ptrace. If desired, another ptrace request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the request argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at addr is returned. If I and D space are separated, request 1 indicates I space, 2 D space. Addr must be even. The child must be stopped. The input data is ignored.

3 The word of the system's per-process data area corresponding to addr is returned. Addr must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the user structure in the system.

4,5 The given data is written at the word in the process's address space corresponding to addr, which must be even. No useful value is returned. If I and D space are
separated, request 4 indicates I space, 5 D space.
Attempts to write in pure procedure fail if another pro-
cess is executing the same file.

6 The process's system data is written, as it is read with
request 3. Only a few locations can be written in this
way: the general registers, the floating point status
and registers, and certain bits of the processor status
word.

7 The data argument is taken as a signal number and the
child's execution continues at location addr as if it
had incurred that signal. Normally the signal number
will be either 0 to indicate that the signal that caused
the stop should be ignored, or that value fetched out of
the process's image indicating which signal caused the
stop. If addr is (int *)1 then execution continues from
where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as
possible after execution of at least one instruction,
execution stops again. The signal number from the stop
is SIGTRAP. (On the PDP-11 and VAX-11 the T-bit is used
and just one instruction is executed; on the Interdata
the stop does not take place until a store instruction
is executed.) This is part of the mechanism for imple-
menting breakpoints.

As indicated, these calls (except for request 0) can be used
only when the subject process has stopped. The wait call is
used to determine when a process stops; in such a case the
'termination' status returned by wait has the value 0177 to
indicate stoppage rather than genuine termination.

To forestall possible fraud, ptrace inhibits the set-user-id
facility on subsequent exec(2) calls. If a traced process
calls exec, it will stop before executing the first instruc-
tion of the new image showing signal SIGTRAP.

On the Interdata 8/32, 'word' means a 32-bit word and 'even'
means 0 mod 4. On a VAX-11, 'word' also means a 32-bit
integer, but the 'even' restriction does not apply.

SEE ALSO
wait(2), signal(2), adb(1)

DIAGNOSTICS
The value -1 is returned if request is invalid, pid is not a
traceable process, addr is out of bounds, or data specifies
an illegal signal number.

**BUGS**

`Ptrace` is unique and arcane; it should be replaced with a special file which can be opened and read and written. The control functions could then be implemented with `ioctl(2)` calls on this file. This would be simpler to understand and have much higher performance.

On the Interdata 8/32, 'as soon as possible' (request 7) means 'as soon as a store instruction has been executed.'

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use 'illegal instruction' signals at a very high rate) could be efficiently debugged.

The error indication, -1, is a legitimate function value; `errno`, see `intro(2)`, can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.
NAME
read - read from file

SYNOPSIS
read(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION
A file descriptor is a word returned from a successful open, creat, dup, or pipe call. Buffer is the location of nbytes contiguous bytes into which the input will be placed. It is not guaranteed that all nbytes bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned.

If the returned value is 0, then end-of-file has been reached.

Unless the reader is ignoring or holding SIGTTIN signals, reads from the control typewriter while not in its process group cause a SIGTTIN signal to be sent to the reader's process group; in the former case an end-of-file is returned.

SEE ALSO
open(2), creat(2), dup(2), pipe(2), vread(2)

DIAGNOSTICS
As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the return value is -1. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous nbytes, file descriptor not that of an input file.

BUGS
It should be possible to call read and have it return immediately without blocking if there is no input available. As a single special case, this is currently done on control terminals when the reading process has requested SIGTINT signals when input arrives (see tty(4)).

Processes which have been orphaned by their parents and have been inherited by init(8) never receive SIGTTIN signals. Instead read returns with an end-of-file indication.
NAME
   setuid, setgid - set user and group ID

SYNOPSIS
   setuid(uid)
   setgid(gid)

DESCRIPTION
   The user ID (group ID) of the current process is set to the argument. Both the effective and the real ID are set. These calls are only permitted to the super-user or if the argument is the real or effective ID.

SEE ALSO
   getuid(2)

DIAGNOSTICS
   Zero is returned if the user (group) ID is set; -1 is returned otherwise.
NAME
signal - catch or ignore signals

SYNOPSIS
#include <signal.h>

(*signal(sig, func))()
void (*func)();

DESCRIPTION
N.B.: The system currently supports two signal implementations. The one described here is standard in version 7 UNIX systems, and is retained for backward compatibility. The one described in sigsys(2) as supplemented by sigset(3) provides for the needs of the job control mechanisms used by csh(1), and corrects the bugs in this older implementation of signals, allowing programs which process interrupts to be written reliably.

A signal is generated by some abnormal event, initiated either by user at a terminal (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals cause termination of the receiving process, but a signal call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals with names as in the include file.

SIGHUP 1 hangup
SIGINT  2 interrupt
SIGQUIT 3* quit
SIGILL  4* illegal instruction (not reset when caught)
SIGTRAP 5* trace trap (not reset when caught)
SIGIOT  6* IOT instruction
SIGEMT  7* EMT instruction
SIGFPE  8* floating point exception
SIGKILL 9 kill (cannot be caught or ignored)
SIGBUS 10* bus error
SIGSEGV 11* segmentation violation
SIGSYS 12* bad argument to system call
SIGPIPE 13 write on a pipe with no one to read it
SIGALRM 14 alarm clock
SIGTERM 15 software termination signal
16 unassigned

N.B.: There are actually more signals; see sigsys(2); the signals listed here are those of standard version 7.

The starred signals in the list above cause a core image if not caught or ignored.
If `func` is SIG_DFL, the default action for signal `sig` is reinstated; this default is termination, sometimes with a core image. If `func` is SIG_IGN the signal is ignored. Otherwise when the signal occurs `func` will be called with the signal number as argument. A return from the function will continue the process at the point it was interrupted.

Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another `signal` call.

If, when using this (older) signal interface, a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during an `ioctl`, `read`, or `write(2)` on a slow device (like a terminal; but not a file); and during `pause` or `wait(2)`. When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catchings takes place, it will appear that the system call returned an error status. The user's program may then, if it wishes, re-execute the call.

The value of `signal` is the previous (or initial) value of `func` for the particular signal.

After a `fork(2)` the child inherits all signals. `Exec(2)` resets all caught signals to default action.

If a process is using the mechanisms of `sigsys(2)` and `sigset(3)` then many of these calls are automatically restarted (See `sigsys(2)` and `jobs(3)` for details).

SEE ALSO
sigsys(2), kill(1), kill(2), ptrace(2), setjmp(3), sigset(3)

DIAGNOSTICS
The value (int)-1 is returned if the given signal is out of range.

BUGS
The traps should be distinguishable by extra arguments to the signal handler, and all hardware supplied parameters should be made available to the signal routine.

If a repeated signal arrives before the last one can be reset, there is no chance to catch it (however this is not true if you use `sigsys(2)` and `sigset(3)`).

The type specification of the routine and its `func` argument are problematical.
NAME
stat, fstat - get file status

SYNOPSIS
#include <sys/types.h>
#include <sys/stat.h>

stat(name, buf)
char *name;
struct stat *buf;

fstat(fildes, buf)
struct stat *buf;

DESCRIPTION
Stat obtains detailed information about a named file. Fstat obtains the same information about an open file known by the file descriptor from a successful open, creat, dup or pipe(2) call.

Name points to a null-terminated string naming a file; buf is the address of a buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be searchable. The layout of the structure pointed to by buf as defined in <stat.h> is given below. St_mode is encoded according to the 'define' statements.

struct stat
{
    dev_t st_dev;
    ino_t st_ino;
    unsigned short st_mode;
    short st_nlink;
    short st_uid;
    short st_gid;
    dev_t st_rdev;
    off_t st_size;
    time_t st_atime;
    time_t st_mtime;
    time_t st_ctime;
};
The mode bits \(0000070\) and \(0000007\) encode group and others permissions (see \texttt{chmod(2)}). The defined types, \texttt{ino_t, off_t, time_t, name}, various width integer values; \texttt{dev_t} encodes major and minor device numbers; their exact definitions are in the include file \texttt{<sys/types.h>} (see \texttt{types(5)}).

When \texttt{fildes} is associated with a pipe, \texttt{fstat} reports an ordinary file with an i-node number, restricted permissions, and a not necessarily meaningful length.

\texttt{st_atime} is the file was last read. For reasons of efficiency, it is not set when a directory is searched, although this would be more logical. \texttt{st_mtime} is the time the file was last written or created. It is not set by changes of owner, group, link count, or mode. \texttt{st_ctime} is set both by writing and changing the i-node.

\textbf{SEE ALSO}
\texttt{ls(1), filsys(5)}

\textbf{DIAGNOSTICS}
Zero is returned if a status is available; \(-1\) if the file cannot be found.
NAME
stime - set time

SYNOPSIS
stime(tp)
long *tp;

DESCRIPTION
Stime sets the system's idea of the time and date. Time,
pointed to by tp, is measured in seconds from 0000 GMT Jan
1, 1970. Only the super-user may use this call.

SEE ALSO
date(1), time(2), ctime(3)

DIAGNOSTICS
Zero is returned if the time was set; -1 if user is not the
super-user.
NAME
sync - update super-block

SYNOPSIS
sync()

DESCRIPTION
sync causes all information in core memory that should be on
disk to be written out. This includes modified super
blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system,
for example icheck, df, etc. It is mandatory before a boot.

SEE ALSO
sync(1), update(8)

BUGS
The writing, although scheduled, is not necessarily complete
upon return from sync.
NAME
time, ftime - get date and time

SYNOPSIS
long time(0)

long time(tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;

DESCRIPTION
Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If tloc is nonnull, the return value is also stored in the place to which tloc points.

The ftime entry fills in a structure pointed to by its argument, as defined by <sys/timeb.h>:

*/
* Structure returned by ftime system call
*
* jam 810817
*/
struct timeb {
    time_t time;
    unsigned short millitm;
    short timezone;
    short dstflag;
};

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.
SEE ALSO
date(1), stime(2), ctime(3)
NAME
	times - get process times

SYNOPSIS
	#include <sys/types.h>
	#include <sys/times.h>

times(buffer)
	struct tms *buffer;

DESCRIPTION
	Times returns time-accounting information for the current
	process and for the terminated child processes of the

current process. All times are in 1/Hz seconds, where HZ is

either 50 or 60 depending on your locality.

This is the structure returned by times:

	/* Structure returned by times() */
	struct tms {
		time_t tms_utime;     /* user time */
		time_t tms_stime;     /* system time */
		time_t tms_cutime;    /* user time, children */
		time_t tms_cstime;    /* system time, children */
	};

The children times are the sum of the children's process

times and their children's times.

SEE ALSO
	time(1), time(2), vtimes(2)
NAME
umask - set file creation mode mask

SYNOPSIS
umask(complmode)

DESCRIPTION
Umask sets a mask used whenever a file is created by creat(2) or mknod(2): the actual mode (see chmod(2)) of the newly-created file is the logical and of the given mode and the complement of the argument. Only the low-order 9 bits of the mask (the protection bits) participate. In other words, the mask shows the bits to be turned off when files are created.

The previous value of the mask is returned by the call. The value is initially 022 (write access for owner only). The mask is inherited by child processes.

SEE ALSO
creat(2), mknod(2), chmod(2)
NAME
unlink - remove directory entry

SYNOPSIS
unlink(name)
char *name;

DESCRIPTION
Name points to a null-terminated string. Unlink removes the entry for the file pointed to by name from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO
rm(l), link(2)

DIAGNOSTICS
Zero is normally returned; -1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).
NAME
utime - set file times

SYNOPSIS
#include <sys/types.h>

utime(file, timep)
char *file;
time_t timep[2];

DESCRIPTION
The utime call uses the 'accessed' and 'updated' times in that order from the timep vector to set the corresponding recorded times for file.

The caller must be the owner of the file or the super-user. The 'inode-changed' time of the file is set to the current time.

SEE ALSO
stat (2)
NAME
write - write on a file

SYNOPSIS
write(fildes, buffer, nbytes)
    char *buffer;

DESCRIPTION
A file descriptor is a word returned from a successful open, creat, dup, or pipe(2) call.

Buffer is the address of nbytes contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 1024 characters long and begin on a 1024-byte boundary in the file are more efficient than any others.

Writes to the control terminal by a process which is not in the process group of the terminal and which is not ignoring or holding SIGTTOU signals cause the writer's process group to receive a SIGTTOU signal (See jobs(3) and the description of the LSTOP option in tty(4) for details).

On some systems write clears the set-user-id bit on a file. This prevents penetration of system security by a user who "captures" a writeable set-user-id file owned by the superuser.

SEE ALSO
creat(2), open(2), pipe(2)

DIAGNOSTICS
Returns -1 on error: bad descriptor, buffer address, or count; physical I/O errors.

BUGS
It would be nice to be able to call write and have the call return with an error indication if there was no buffer space for the written data, rather than blocking the process.

Processes which have been orphaned by their parents and have been inherited by init(8) never receive SIGTTOU signals. Output by such a process is permitted even when they are not in the process group of the control terminal.
Part 2 Library Functions

These procedures provide the runtime support for the C language. This support includes various methods of I/O, a variety of mathematical functions (including the transcendental functions), and a general set of subroutines to facilitate programming.
NAME
intro - introduction to library functions

SYNOPSIS
#include <stdio.h>
#include <math.h>

DESCRIPTION
This section describes functions that may be found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in section 2. Functions are divided into various libraries distinguished by the section number at the top of the page:

(3) These functions, together with those of section 2 and those marked (3S), constitute library libc, which is automatically loaded by the C compiler cc(1) and the Fortran compiler f77(1). The link editor ld(1) searches this library under the `-lc' option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3J) These functions are part of the job control facilities, contained in the library "$ ./jobs -ljobs". The job control facilities are outlined in jobs(3).

(3M) These functions constitute the math library, libm. They are automatically loaded as needed by the Fortran compiler f77(1). The link editor searches this library under the `-lm' option. Declarations for these functions may be obtained from the include file <math.h>.

(3S) These functions constitute the `standard I/O package', see stdio(3). These functions are in the library libc already mentioned. Declarations for these functions may be obtained from the include file <stdio.h>.

(3X) Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages.

FILES
/lib/libc.a
/lib/libm.a, /usr/lib/libm.a (one or the other)

SEE ALSO
stdio(3), nm(1), ld(1), cc(1), f77(1), intro(2)
DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable errno (see intro(2)) is set to the value EDOM or ERANGE. The values of EDOM and ERANGE are defined in the include file <math.h>.
NAME
 abort - generate a fault

DESCRIPTION
 Abort executes an instruction which is illegal in user mode. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

SEE ALSO
 adb(1), signal(2), exit(2)

DIAGNOSTICS
 Usually 'IOT trap - core dumped' from the shell.
NAME
    abs - integer absolute value

SYNOPSIS
    abs(i)
    int i;

DESCRIPTION
    abs returns the absolute value of its integer operand.

SEE ALSO
    floor(3) for fabs

BUGS
    You get what the hardware gives on the smallest integer.
NAME
atof, atoi, atol - convert ASCII to numbers

SYNOPSIS

double atof(nptr)
  char *nptr;

atoi(nptr)
  char *nptr;

long atol(nptr)
  char *nptr;

DESCRIPTION
These functions convert a string pointed to by nptr to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi and atol recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

SEE ALSO
  scanf(3)

BUGS
There are no provisions for overflow.
NAME
crypt, setkey, encrypt - DES encryption

SYNOPSIS
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;

DESCRIPTION
Crypt is the password encryption routine. It is based on
the NBS Data Encryption Standard, with variations intended
(among other things) to frustrate use of hardware implemen-
tations of the DES for key search.

The first argument to crypt is a user's typed password. The
second is a 2-character string chosen from the set [a-zA-
Z0-9./]. The salt string is used to perturb the DES algo-
rithm in one of 4096 different ways, after which the pass-
word is used as the key to encrypt repeatedly a constant
string. The returned value points to the encrypted pass-
word, in the same alphabet as the salt. The first two char-
acters are the salt itself.

The other entries provide (rather primitive) access to the
actual DES algorithm. The argument of setkey is a character
array of length 64 containing only the characters with
numerical value 0 and 1. If this string is divided into
groups of 8, the low-order bit in each group is ignored,
leading to a 56-bit key which is set into the machine.

The argument to the encrypt entry is likewise a character
array of length 64 containing 0's and 1's. The argument
array is modified in place to a similar array representing
the bits of the argument after having been subjected to the
DES algorithm using the key set by setkey. If edflag is 0,
the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO
passwd(1), passwd(5), login(1), getpass(3)

BUGS
The return value points to static data whose content is
overwritten by each call.
NAME
ctime, localtime, gmtime, asctime, timezone - convert date
and time to ASCII

SYNOPSIS
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)

DESCRIPTION
Ctime converts a time pointed to by clock such as returned
by time(2) into ASCII and returns a pointer to a 26-
character string in the following form. All the fields have
constant width.

Sun Sep 16 01:03:52 1973

Localtime and gmtime return pointers to structures contain­
ing the broken-down time. localtime corrects for the time
zone and possible daylight savings time; gmtime converts
directly to GMT, which is the time UNIX uses. Asctime con­
verts a broken-down time to ASCII and returns a pointer to a
26-character string.

The structure declaration from the include file is:

struct tm { /* see ctime(3) */
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
}

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the standard U.S.A. daylight saving time adjustment is appropriate. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

Timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Saving version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g. in Afghanistan timezone(-(60*4+30), 0) is appropriate because it is 4:30 ahead of GMT and the string GMT+4:30 is produced.

SEE ALSO
time(2)

BUGS
The return values point to static data whose content is overwritten by each call.
NAME
  isalpha, isupper, islower, isdigit, isalnum, isspace,
ispunct, isprint, iscntrl, isascii - character classification

SYNOPSIS
  #include <ctype.h>

  isalpha(c)
  ...

DESCRIPTION
  These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. isascii is defined on all integer values; the rest are defined only where isascii is true and on the single non-ASCII value EOF (see stdio(3)).

  isalpha  c is a letter
  isupper  c is an upper case letter
  islower  c is a lower case letter
  isdigit  c is a digit
  isalnum  c is an alphanumeric character
  isspace  c is a space, tab, carriage return, newline, or formfeed
  ispunct  c is a punctuation character (neither control nor alphanumeric)
  isprint  c is a printing character, code 040(8)
           (space) through 0176 (tilde)
  iscntrl  c is a delete character (0177) or ordinary control character (less than 040).
  isascii  c is an ASCII character, code less than 0200

SEE ALSO
  ascii(7)
NAME
ecvt, fcvt, gcvt - output conversion

SYNOPSIS

char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;

DESCRIPTION

Ecvt converts the value to a null-terminated string of ndigit ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through decpt (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by sign is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to ecvt, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by ndigits.

Gcvt converts the value to a null-terminated ASCII string in buf and returns a pointer to buf. It attempts to produce ndigit significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3)

BUGS

The return values point to static data whose content is overwritten by each call.
NAME
exp, log, logl0, pow, sqrt - exponential, logarithm, power, square root

SYNOPSIS
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double logl0(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;

DESCRIPTION
Exp returns the exponential function of x.
Log returns the natural logarithm of x; logl0 returns the base 10 logarithm.
Pow returns x^y.
Sqrt returns the square root of x.

SEE ALSO
hypot(3), sinh(3), intro(2)

DIAGNOSTICS
Exp and pow return a huge value when the correct value would overflow; errno is set to ERANGE. Pow returns 0 and sets errno to EDOM when the second argument is negative and non-integral and when both arguments are 0.

Log returns 0 when x is zero or negative; errno is set to EDOM.

Sqrt returns 0 when x is negative; errno is set to EDOM.
NAME
fclose, fflush - close or flush a stream

SYNOPSIS
#include <stdio.h>

close(stream)
FILE *stream;

fflush(stream)
FILE *stream;

DESCRIPTION
fclose causes any buffers for the named stream to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

fclose is performed automatically upon calling exit(2).

fflush causes any buffered data for the named output stream to be written to that file. The stream remains open.

SEE ALSO
close(2), fopen(3), setbuf(3)

DIAGNOSTICS
These routines return EOF if stream is not associated with an output file, or if buffered data cannot be transferred to that file.
NAME
f abs, floor, ceil - absolute value, floor, ceiling functions

SYNOPSIS
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;

DESCRIPTION
fabs returns the absolute value \(|x|\).

Floor returns the largest integer not greater than \(x\).

Ceil returns the smallest integer not less than \(x\).

SEE ALSO
abs(3) S=1 .ds ]D Fortune Operating System S=1 .ds ]E
Development Set S=2 .ds ]D System Routines S=3 .ds ]D
Library Functions S=5 .ds ]D File Formats
NAME
fopen, freopen, fdopen — open a stream

SYNOPSIS
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen(fildes, type)
char *type;

DESCRIPTION
fopen opens the file named by filename and associates a stream with it. fopen returns a pointer to be used to identify the stream in subsequent operations.

Type is a character string having one of the following values:

"r" open for reading

"w" create for writing

"a" append: open for writing at end of file, or create for writing

In addition, each type may be followed by a '+' to have the file opened for reading and writing. "r+" positions the stream at the beginning of the file, "w+" creates or truncates it, and "a+" positions it at the end. Both reads and writes may be used on read/write streams, with the limitation that an fseek, rewind, or reading an end-of-file must be used between a read and a write or vice-versa.

freopen substitutes the named file in place of the open stream. It returns the original value of stream. The original stream is closed.

freopen is typically used to attach the preopened constant names, stdin, stdout, stderr, to specified files.

fdopen associates a stream with a file descriptor obtained from open, dup, creat, or pipe(2). The type of the stream must agree with the mode of the open file.
SEE ALSO
    open(2), fclose(3)

DIAGNOSTICS
    Fopen and freopen return the pointer NULL if filename cannot
    be accessed.

BUGS
    Fdopen is not portable to systems other than UNIX.

    The read/write types do not exist on all systems. Those
    systems without read/write modes will probably treat the
    type as if the '+' was not present.
NAME
fread, fwrite - buffered binary input/output

SYNOPSIS
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;

fwrite(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;

DESCRIPTION
Fread reads, into a block beginning at ptr, nitems of data of the type of *ptr from the named input stream. It returns the number of items actually read.

If stream is stdin and the standard output is line buffered, then any partial output line will be flushed before any call to read(2) to satisfy the fread.

Fwrite appends at most nitems of data of the type of *ptr beginning at ptr to the named output stream. It returns the number of items actually written.

SEE ALSO
read(2), write(2), fopen(3), getc(3), putc(3), gets(3), puts(3), printf(3), scanf(3)

DIAGNOSTICS
Fread and fwrite return 0 upon end of file or error.

BUGS
NAME
frexp, ldexp, modf - split into mantissa and exponent

SYNOPSIS

double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;

DESCRIPTION
Frexp returns the mantissa of a double value as a double quantity, x, of magnitude less than 1 and stores an integer n such that value = x * 2^n indirectly through eptr.

Ldexp returns the quantity value * 2^n.

Modf returns the positive fractional part of value and stores the integer part indirectly through iptr.

Fortune Operating System Development Set
System Routines Library Functions File Formats
NAME
fseek, ftell, rewind - reposition a stream

SYNOPSIS
#include <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(stream)

DESCRIPTION
fseek sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes from the beginning, the current position, or the end of the file, according as ptrname has the value 0, 1, or 2.

fseek undoes any effects of ungetc(3).

ftell returns the current value of the offset relative to the beginning of the file associated with the named stream. It is measured in bytes on UNIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an offset for fseek.

rewind(stream) is equivalent to fseek(stream, 0L, 0).

SEE ALSO
lseek(2), fopen(3)

DIAGNOSTICS
NAME
 gamma - log gamma function

SYNOPSIS
#include <math.h>

double gamma(x)
double x;

DESCRIPTION
 Gamma returns ln |G(|x|)|. The sign of G(|x|) is returned in the external integer signgam. The following C program might be used to calculate G:

    y = gamma(x);
    if (y > 88.0)
      error();
    y = exp(y);
    if(signgam)
      y = -y;

DIAGNOSTICS
A huge value is returned for negative integer arguments.

BUGS
There should be a positive indication of error.
NAME
  getc, getchar, fgetc, getw - get character or word from stream

SYNOPSIS
  #include <stdio.h>

  int getc(stream)
  FILE *stream;

  int getchar()

  int fgetc(stream)
  FILE *stream;

  int getw(stream)
  FILE *stream;

DESCRIPTION
Getc returns the next character from the named input stream.

Getchar() is identical to getc(stdin).

Fgetc behaves like getc, but is a genuine function, not a macro; it may be used to save object text.

Getw returns the next word (32-bit integer on a VAX-ll) from the named input stream. It returns the constant EOF upon end of file or error, but since that is a good integer value, feof and ferror(3) should be used to check the success of getw. Getw assumes no special alignment in the file.

SEE ALSO
  fopen(3), putc(3), gets(3), scanf(3), fread(3), ungetc(3)

DIAGNOSTICS
These functions return the integer constant EOF at end of file or upon read error.

A stop with message, 'Reading bad file', means an attempt has been made to read from a stream that has not been opened for reading by fopen.

BUGS
The end-of-file return from getchar is incompatible with that in UNIX editions 1-6.

Because it is implemented as a macro, getc treats a stream argument with side effects incorrectly. In particular, 'getc(*f++);' doesn't work sensibly.
NAME
    getenv - value for environment name

SYNOPSIS
    char *getenv(name)
    char *name;

DESCRIPTION
    Getenv searches the environment list (see environ(5)) for a
    string of the form name=value and returns value if such a
    string is present, otherwise 0 (NULL).

SEE ALSO
    environ(5), exec(2)
NAME
getgrent, getgrgid, getgrnam, setgrent, endgrent - get group file entry

SYNOPSIS
#include <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

setgrent()

endgrent()

DESCRIPTION
Getgrent, getgrgid and getgrnam each return pointers to an object with the following structure containing the broken-out fields of a line in the group file.

struct group { /* see getgrent(3) */
    char *gr_name;
    char *gr_passwd;
    int gr_gid;
    char **gr_mem;
};

The members of this structure are:

gr_name The name of the group.
gr_passwd The encrypted password of the group.
gr_gid The numerical group-ID.
gr_mem Null-terminated vector of pointers to the individual member names.

Getgrent simply reads the next line while getgrgid and getgrnam search until a matching gid or name is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.
A call to `setgrent` has the effect of rewinding the group file to allow repeated searches. `Endgrent` may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

`getlogin(3), getpwent(3), group(5)`

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.
NAME
getlogin - get login name

SYNOPSIS
cchar *getlogin()

DESCRIPTION
Getlogin returns a pointer to the login name as found in
/etc/utmp. It may be used in conjunction with getpwnam to
locate the correct password file entry when the same userid
is shared by several login names.

If getlogin is called within a process that is not attached
to a typewriter, it returns NULL. The correct procedure for
determining the login name is to first call getlogin and if
it fails, to call getpwuid.

FILES
/etc/utmp

SEE ALSO
getpwent(3), getgrent(3), utmp(5)

DIAGNOSTICS
Returns NULL (0) if name not found.

BUGS
The return values point to static data whose content is
overwritten by each call.
NAME
   getpass - read a password

SYNOPSIS
   char *getpass(prompt)
   char *prompt;

DESCRIPTION
   Getpass reads a password from the file /dev/tty, or if that
cannot be opened, from the standard input, after prompting
with the null-terminated string prompt and disabling echo-
ing. A pointer is returned to a null-terminated string of
at most 8 characters.

FILES
   /dev/tty

SEE ALSO
   crypt(3)

BUGS
   The return value points to static data whose content is
   overwritten by each call.
NAME
getpw - get name from uid

SYNOPSIS
getpw(uid, buf)
char *buf;

DESCRIPTION
Getpw searches the password file for the (numerical) uid, and fills in buf with the corresponding line; it returns non-zero if uid could not be found. The line is null-terminated.

FILES
/etc/passwd

SEE ALSO
getpwent(3), passwd(5)

DIAGNOSTICS
Non-zero return on error.
NAME
getpwent, getpwuid, getpwnam, setpwent, endpwent - get password file entry

SYNOPSIS
#include <pwd.h>

struct passwd *getpwent()
struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)  
char *name;

int setpwent()
int endpwent()

DESCRIPTION
Getpwent, getpwuid and getpwnam each return a pointer to an object with the following structure containing the broken-out fields of a line in the password file.

```
struct passwd { /* see getpwent(3) */
    char *pw_name;
    char *pw_passwd;
    int pw_uid;
    int pw_gid;
    int pw_quota;
    char *pw_comment;
    char *pw_gecos;
    char *pw_dir;
    char *pw_shell;
};
```

The fields pw_quota and pw_comment are unused; the others have meanings described in passwd(5).

Getpwent reads the next line (opening the file if necessary); setpwent rewinds the file; endpwent closes it.

Getpwuid and getpwnam search from the beginning until a matching uid or name is found (or until EOF is encountered).
FILES
/etc/passwd

SEE ALSO
getlogin(3), getgrent(3), passwd(5)

DIAGNOSTICS
Null pointer (0) returned on EOF or error.

BUGS
All information is contained in a static area so it must be copied if it is to be saved.
NAME

gets, fgets - get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets(s)
char *s;

cchar *fgets(s, n, stream)
char *s;
FILE *stream;

DESCRIPTION

Gets reads a string into s from the standard input stream stdin. The string is terminated by a newline character, which is replaced in s by a null character. Gets returns its argument.

Fgets reads n-1 characters, or up to a newline character, whichever comes first, from the stream into the string s. The last character read into s is followed by a null character. Fgets returns its first argument.

SEE ALSO

puts(3), getc(3), scanf(3), fread(3), ferror(3)

DIAGNOSTICS

Gets and fgets return the constant pointer NULL upon end of file or error.

BUGS

Gets deletes a newline, fgets keeps it, all in the name of backward compatibility. M=1 .ds ID Fortune Operating System M=1 .ds J E Development Set M=2 .ds ID System Routines M=3 .ds ID Library Functions M=5 .ds ID File Formats
NAME
  hypot, cabs - Euclidean distance

SYNOPSIS
#include <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y; } z;

DESCRIPTION
  Hypot and cabs return
  \( \sqrt{x^2 + y^2} \),
  taking precautions against unwarranted overflows.

SEE ALSO
  exp(3) for sqrt M=1 .ds ]D Fortune Operating System M=1
NAME
j0, j1, jn, y0, y1, yn - bessel functions

SYNOPSIS
#include <math.h>

double j0(x)

double x;

double j1(x)

double x;

double jn(n, x)

double x;

double y0(x)

double x;

double y1(x)

double x;

double yn(n, x)

double x;

DESCRIPTION
These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS
Negative arguments cause y0, y1, and yn to return a huge negative value and set errno to EDOM.
NAME
l3tol, lto13 - convert between 3-byte integers and long integers

SYNOPSIS
l3tol(lp, cp, n)
long *lp;
char *cp;

lto13(cp, lp, n)
char *cp;
long *lp;

DESCRIPTION
l3tol converts a list of n three-byte integers packed into a character string pointed to by cp into a list of long integers pointed to by lp.

lto13 performs the reverse conversion from long integers (lp) to three-byte integers (cp).

These functions are useful for file-system maintenance where the i-numbers are three bytes long.

SEE ALSO
filsys(5)
NAME
malloc, free, realloc, calloc - main memory allocator

SYNOPSIS
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;

DESCRIPTION
Malloc and free provide a simple general-purpose memory allocation package. Malloc returns a pointer to a block of at least size bytes beginning on a word boundary.

The argument to free is a pointer to a block previously allocated by malloc; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by malloc is overrun or if some random number is handed to free.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls sbrk (see break(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Realloc also works if ptr points to a block freed since the last call of malloc, realloc or calloc; thus sequences of free, malloc and realloc can exploit the search strategy of malloc to do storage compaction.

Calloc allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.
Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

**DIAGNOSTICS**

`malloc`, `realloc` and `callcc` return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. `malloc` may be recompiled to check the arena very stringently on every transaction; see the source code.

**BUGS**

When `realloc` returns 0, the block pointed to by `ptr` may be destroyed.

The current incarnation of the allocator is unsuitable for direct use in a large virtual environment where many small blocks are to be kept, since it keeps all allocated and freed blocks on a single circular list. Just before more memory is allocated, all allocated and freed blocks are referenced; this can cause a huge number of page faults.
NAME
  mktemp - make a unique file name

SYNOPSIS
  char *mktemp(template)
  char *template;

DESCRIPTION
  Mktemp replaces template by a unique file name, and returns
  the address of the template. The template should look like
  a file name with six trailing X's, which will be replaced
  with the current process id and a unique letter.

SEE ALSO.
  getpid(2)
NAME

monitor - prepare execution profile

SYNOPSIS

monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
short buffer[];

DESCRIPTION

An executable program created by `cc -p' automatically includes calls for monitor with default parameters; monitor needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to profil(2). Lowpc and highpc are the addresses of two functions; buffer is the address of a (user supplied) array of bufsize short integers. Monitor arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of lowpc and the highest is just below highpc. At most nfunc call counts can be kept; only calls of functions compiled with the profiling option -p of cc(1) are recorded. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

    extern etext();
    ...
    monitor((int) 2, etext, buf, bufsize, nfunc);

Etext lies just above all the program text, see end(3).

To stop execution monitoring and write the results on the file mon.out, use

    monitor(0);

then prof(1) can be used to examine the results.

FILES

mon.out

SEE ALSO

prof(1), profil(2), cc(1)
NAME
nlist - get entries from name list

SYNOPSIS
#include <nlist.h>
nlist(filename, nl)
char *filename;
struct nlist nl[];

DESCRIPTION
Nlist examines the name list in the given executable output
file and selectively extracts a list of values. The name
list consists of an array of structures containing names,
types and values. The list is terminated with a null name.
Each name is looked up in the name list of the file. If the
name is found, the type and value of the name are inserted
in the next two fields. If the name is not found, both
entries are set to 0. See a.out(5) for the structure
declaration.

This subroutine is useful for examining the system name list
kept in the file /vmunix. In this way programs can obtain
system addresses that are up to date.

SEE ALSO
a.out(5)

DIAGNOSTICS
All type entries are set to 0 if the file cannot be found or
if it is not a valid namelist.

BUGS
On other versions of UNIX you must include <a.out.h> rather
than <nlist.h>; this is unfortunate, but <a.out.h> can't be
used on the VAX because it contains a union which can't be
initialized.
NAME
 perror, sys_errlist, sys_nerr - system error messages

SYNOPSIS
 perror(s)
   char *s;

   int sys_nerr;
   char *sys_errlist[];

DESCRIPTION
 Perror produces a short error message on the standard error
 file describing the last error encountered during a call to
 the system from a C program. First the argument string s is
 printed, then a colon, then the message and a new-line.
 Most usefully, the argument string is the name of the pro­
 gram which incurred the error. The error number is taken
 from the external variable errno (see intro(2)), which is
 set when errors occur but not cleared when non-erroneous
 calls are made.

 To simplify variant formatting of messages, the vector of
 message strings sys_errlist is provided; errno can be used
 as an index in this table to get the message string without
 the newline. Sys_nerr is the number of messages provided
 for in the table; it should be checked because new error
 codes may be added to the system before they are added to
 the table.

SEE ALSO
 intro(2) S=1 .ds ]D Fortune Operating System S=1 .ds ]E
 Development Set S=2 .ds ]D System Routines S=3 .ds ]D
 Library Functions S=5 .ds ]D File Formats
NAME
popen, pclose - initiate I/O to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;

DESCRIPTION
The arguments to popen are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by popen should be closed by pclose, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

SEE ALSO
pipe(2), fopen(3), fclose(3), system(3), wait(2)

DIAGNOSTICS
Popen returns a null pointer if files or processes cannot be created, or the Shell cannot be accessed.

Pclose returns -1 if stream is not associated with a 'popen' command.

BUGS
Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with fflush, see fcloae(3).
NAME
printf, fprintf, sprintf — formatted output conversion

SYNOPSIS
#include <stdio.h>

printf(format [, arg ] ... )
char *format;

fprintf(stream, format [, arg ] ... )
FILE *stream;
char *format;

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

DESCRIPTION
Printf places output on the standard output stream stdout.
Fprintf places output on the named output stream. Sprintf places 'output' in the string s, followed by the character '\0'.

Each of these functions converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive arg printf.

Each conversion specification is introduced by the character %. Following the %, there may be:

- an optional minus sign '-' which specifies left adjustment of the converted value in the indicated field;

- an optional digit string specifying a field width; if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;

- an optional period '.' which serves to separate the field width from the next digit string;

- an optional digit string specifying a precision which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- the character \( l \) specifying that a following \( d, o, x, \) or \( u \) corresponds to a long integer \( \text{arg} \). (A capitalized conversion code accomplishes the same thing.)

- a character which indicates the type of conversion to be applied.

A field width or precision may be `*` instead of a digit string. In this case an integer \( \text{arg} \) supplies the field width or precision.

The conversion characters and their meanings are

- `d` The integer \( \text{arg} \) is converted to decimal, octal, or hexadecimal notation respectively.

- `f` The float or double \( \text{arg} \) is converted to decimal notation in the style `\[-\]ddd.ddd` where the number of \( d \)'s after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

- `e` The float or double \( \text{arg} \) is converted in the style `\[-\]d.ddde±dd` where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

- `g` The float or double \( \text{arg} \) is printed in style `d`, in style `f`, or in style `e`, whichever gives full precision in minimum space.

- `c` The character \( \text{arg} \) is printed.

- `s` \( \text{Arg} \) is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

- `u` The unsigned integer \( \text{arg} \) is converted to decimal and printed (the result will be in the range 0 through \text{MAXUINT}, where \text{MAXUINT} equals 4294967295 on a VAX-11 and 65536 on a PDP-11).

- `%` Print a `\%`; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the
specified field width exceeds the actual width. Characters generated by `printf` are printed by `putc(3)`.

**Examples**

To print a date and time in the form 'Sunday, July 3, 10:02', where `weekday` and `month` are pointers to null-terminated strings:

```c
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

To print pi to 5 decimals:

```c
printf("pi = %.5f", 4*atan(1.0));
```

**SEE ALSO**

`putc(3), scanf(3), ecvt(3)`

**BUGS**

NAME
putc, putchar, fputc, putw - put character or word on a stream

SYNOPSIS
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putc(c)

putchar(c)

fputc(c, stream)
FILE *stream;

putw(w, stream)
FILE *stream;

DESCRIPTION
Putc appends the character c to the named output stream. It returns the character written.

Putchar(c) is defined as putc(c, stdout).

Fputc behaves like putc, but is a genuine function rather than a macro. It may be used to save on object text.

Putw appends word (i.e. int) w to the output stream. It returns the word written. Putw neither assumes nor causes special alignment in the file.

The standard stream stdout is normally buffered if and only if the output does not refer to a terminal; this default may be changed by setbuf(3). The standard stream stderr is by default unbuffered unconditionally, but use of freopen (see fopen(3)) will cause it to become buffered; setbuf, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. Fflush (see fclose(3)) may be used to force the block out early.

SEE ALSO
fopen(3), fclose(3), getc(3), puts(3), printf(3), fread(3)

DIAGNOSTICS
These functions return the constant EOF upon error. Since this is a good integer, ferror(3) should be used to detect putw errors.
BUGS

Because it is implemented as a macro, _putc_ treats a _stream_ argument with side effects improperly. In particular

`putc(c, *f++)';` doesn't work sensibly.

Errors can occur long after the call to _putc_.

Fortune Operating System S=1 .ds ]D
NAME
puts, fputs - put a string on a stream

SYNOPSIS
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;

DESCRIPTION
.puts copies the null-terminated string s to the standard output stream stdout and appends a newline character.

.fputs copies the null-terminated string s to the named output stream.

Neither routine copies the terminal null character.

SEE ALSO
fopen(3), gets(3), putc(3), printf(3), ferror(3)
fread(3) for fwrite

BUGS
.puts appends a newline, fputs does not, all in the name of backward compatibility.
NAME
qsort - quicker sort

SYNOPSIS
qsort(base, nel, width, compar)
char *base;
int (*compar)();

DESCRIPTION
qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO
sort(1)
NAME
rand, srand - random number generator

SYNOPSIS
srand(seed)
int seed;

rand()

DESCRIPTION
Rand uses a multiplicative congruential random number generator with period 28329 to return successive pseudo-random numbers in the range from 0 to 28319-1.

The generator is reinitialized by calling srand with 1 as argument. It can be set to a random starting point by calling srand with whatever you like as argument.
NAME
scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS
#include <stdio.h>

scanf(format [, pointer] ...) char *format;

fscanf(stream, format [, pointer] ...) FILE *stream;
char *format;

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

DESCRIPTION
Scanf reads from the standard input stream stdin. Fscanf reads from the named input stream. Sscanf reads from the character string s. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string format, described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.

2. An ordinary character (not %) which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion charac-

49
ters are legal:
% a single '%' is expected in the input at this point; no assignment is done.
d a decimal integer is expected; the corresponding argument should be an integer pointer.
o an octal integer is expected; the corresponding argument should be an integer pointer.
x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.
c a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try '%ls'. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.
99f7 a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.
[l indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o and x may be capitalized or preceded by l to indicate that a pointer to long rather
than to int is in the argument list. Similarly, the conver-
sion characters e or f may be capitalized or preceded by l
to indicate a pointer to double rather than to float. The
conversion characters d, o and x may be preceded by h to
indicate a pointer to short rather than to int.

The scanf functions return the number of successfully
matched and assigned input items. This can be used to de-
cide how many input items were found. The constant EOF is
returned upon end of input; note that this is different from
0, which means that no conversion was done; if conversion
was intended, it was frustrated by an inappropriate charac-
ter in the input.

For example, the call

```c
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25 54.32E-1 thompson
```

will assign to i the value 25, x the value 5.432, and name
will contain `thompson\0'. Or,

```c
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to i, 789.0 to x, skip `0123', and place the
string `56\0' in name. The next call to getchar will return
`a'.

SEE ALSO
atof(3), getc(3), printf(3)

DIAGNOSTICS
The scanf functions return EOF on end of input, and a short
count for missing or illegal data items.

BUGS
The success of literal matches and suppressed assignments is
not directly determinable.
NAME
setbuf - assign buffering to a stream

SYNOPSIS
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;

DESCRIPTION
Setbuf is used after a stream has been opened but before it is read or written. It causes the character array buf to be used instead of an automatically allocated buffer. If buf is the constant pointer NULL, input/output will be completely unbuffered.

A manifest constant BUFSIZ tells how big an array is needed:

    char buf[BUFSIZ];

A buffer is normally obtained from malloc(3) upon the first getc or putc(3) on the file, except that the standard output is line buffered when directed to a terminal. Other output streams directed to terminals, and the standard error stream stderr are normally not buffered. If the standard output is line buffered, then it is flushed each time data is read from the standard input by read(2).

SEE ALSO
fopen(3), getc(3), putc(3), malloc(3)

BUGS
The standard error stream should be line buffered by default.
NAME

setjmp, longjmp - non-local goto

SYNOPSIS

#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;

DESCRIPTION

These routines are useful for dealing with errors and inter­rupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in env for later use by longjmp. It returns value 0.

Longjmp restores the environment saved by the last call of setjmp. It then returns in such a way that execution con­tinues as if the call of setjmp had just returned the value val to the function that invoked setjmp, which must not it­self have returned in the interim. All accessible data have values as of the time longjmp was called.

SEE ALSO

NAME
sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;

DESCRIPTION
Sin, cos and tan return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

Asin returns the arc sin in the range \(-\frac{\pi}{2}\) to \(\frac{\pi}{2}\).

Acos returns the arc cosine in the range 0 to \(\pi\).

Atan returns the arc tangent of \(x\) in the range \(-\frac{\pi}{2}\) to \(\frac{\pi}{2}\).

Atan2 returns the arc tangent of \(\frac{y}{x}\) in the range \(-\pi\) to \(\pi\).

DIAGNOSTICS
Arguments of magnitude greater than 1 cause asin and acos to return value 0; \texttt{errno} is set to EDOM. The value of tan at its singular points is a huge number, and \texttt{errno} is set to ERANGE.

BUGS
The value of tan for arguments greater than about \(2^{31}\) is garbage.
NAME
sinh, cosh, tanh - hyperbolic functions

SYNOPSIS
#include <math.h>

double sinh(x)

double cosh(x)

double x;

double tanh(x)

double x;

DESCRIPTION
These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS
Sinh and cosh return a huge value of appropriate sign when the correct value would overflow.
NAME
sleep - suspend execution for interval

SYNOPSIS
sleep(seconds)
unsigned seconds;

DESCRIPTION
The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an alarm clock signal and pausing until it occurs. The previous state of this signal is saved and restored. If the sleep time exceeds the time to the alarm signal, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

SEE ALSO
alarm(2), pause(2) S=1 .ds JD Fortune Operating System
S=1 .ds JE Development Set S=2 .ds JD System Routines
S=3 .ds JD Library Functions S=5 .ds JD File Formats
NAME
stdio - standard buffered input/output package

SYNOPSIS
#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

DESCRIPTION
The functions described in Sections 3S constitute an effi-
cient user-level buffering scheme. The in-line macros getc
and putc(3) handle characters quickly. The higher level
routines gets, fgets, scanf, fscanf, fread, puts, fputs,
printf, fprintf, fwrite all use getc and putc; they can be
freely intermixed.

A file with associated buffering is called a stream, and is
declared to be a pointer to a defined type FILE. Fopen(3)
creates certain descriptive data for a stream and returns a
pointer to designate the stream in all further transactions.
There are three normally open streams with constant pointers
declared in the include file and associated with the stan-
dard open files:

stdin standard input file
stdout standard output file
stderr standard error file

A constant 'pointer' NULL (0) designates no stream at all.

An integer constant EOF (-1) is returned upon end of file or
error by integer functions that deal with streams.

Any routine that uses the standard input/output package must
include the header file <stdio.h> of pertinent macro defini-
tions. The functions and constants mentioned in sections
labeled 3S are declared in the include file and need no
further declaration. The constants, and the following
'functions' are implemented as macros; redeclaration of
these names is perilous: getc, getchar, putc, putchar, feof,
ferror, fileno.

SEE ALSO
open(2), close(2), read(2), write(2)

DIAGNOSTICS
The value EOF is returned uniformly to indicate that a FILE
pointer has not been initialized with fopen, input (output)
has been attempted on an output (input) stream, or a FILE
pointer designates corrupt or otherwise unintelligible FILE data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a read(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard i/o routines but use read(2) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to fflush(3) the standard output before going off and computing so that the output will appear.
NAME
strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen,
index, rindex - string operations

SYNOPSIS
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;

DESCRIPTION
These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string s2 to the end of string s1. Strncat copies at most n characters. Both return a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as s1 is lexicographically greater than, equal to, or less than s2. strncmp makes the same comparison but looks at at most n characters.

strcpy copies string s2 to s1, stopping after the null character has been moved. strncpy copies exactly n characters, truncating or null-padding s2; the target may not be null-terminated if the length of s2 is n or more. Both return s1.
**STRING(3) Library Functions STRING(3)**

`strlen` returns the number of non-null characters in `s`.

`index` (or `rindex`) returns a pointer to the first (last) occurrence of character `c` in string `s`, or zero if `c` does not occur in the string.

**BUGS**

`strcmp` uses native character comparison, which is signed on PDP11's and VAX-11's, unsigned on other machines.
NAME
swab - swap bytes

SYNOPSIS
swab(from, to, nbytes)
char *from, *to;

DESCRIPTION
Swab copies nbytes bytes pointed to by from to the position
pointed to by to, exchanging adjacent even and odd bytes.
It is useful for carrying binary data between PDP11's and
other machines. Nbytes should be even.
NAME
  system - issue a shell command

SYNOPSIS
  system(string)
  char *string;

DESCRIPTION
  System causes the string to be given to sh(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO
  popen(3), exec(2), wait(2)

DIAGNOSTICS
  Exit status 127 indicates the shell couldn't be executed.
The file formats describe the structure and conventions of particular UNIX system files. Two examples of these files are the a.out file, which is the file output by the assembler and loader, and the ttys file, which is the file containing the terminal initialization data.
NAME
a.out - assembler and link editor output

SYNOPSIS
#include <a.out.h>

DESCRIPTION
A.out is the output file of the assembler as(1) and the link
editor ld(1). Both programs make a.out executable if there
were no errors and no unresolved external references. Lay­
out information as given in the include file for the VAX-11 is:

/*
 * Header prepended to each a.out file.
 */
struct exec {
  long    a_magic;    /* magic number */
  unsigned a_text;   /* size of text segment */
  unsigned a_data;   /* size of initialized data */
  unsigned a_bss;    /* size of uninitialized data */
  unsigned a_syms;   /* size of symbol table */
  unsigned a_entry;  /* entry point */
  unsigned a_trsize; /* size of text relocation */
  unsigned a_drsiz

#define OMAGIC 0407    /* old impure format */
#define NMAGIC 0410    /* read-only text */
#define ZMAGIC 0413    /* demand load format */

/*
 * Macros which take exec structures as arguments and tell whether
 * the file has a reasonable magic number or offsets to text|symbo
 */
#define N_BADMAG(x)  
  (((x).a_magic)!=OMAGIC && ((x).a_magic)!=NMAGIC && ((x).a_magic)
#define N_TXTOFF(x)  
  ((x).a_magic==ZMAGIC ? 1024 : sizeof (struct exec))
#define N_SYMOFF(x)  
  (N_TXTOFF(x) + (x).a_text+(x).a_data + (x).a_trsize+(x).a
#define N_STROFF(x)  
  (N_SYMOFF(x) + (x).a_syms)

The file has five sections: a header, the program text and
data, relocation information, a symbol table and a string
table (in that order). The last three may be omitted if the
program was loaded with the `-s' option of ld or if the sym­
bols and relocation have been removed by strip(1).
In the header the sizes of each section are given in bytes. The size of the header is not included in any of the other sizes.

When an a.out file is executed, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number in the header is OMAGIC (0407), it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. This is the oldest kind of executable program and is rarely used. If the magic number is NMAGIC (0410) or ZMAGIC (0413), the data segment begins at the first 0 mod 1024 byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. For ZMAGIC format, the text segment begins at a 0 mod 1024 byte boundary in the a.out file, the remaining bytes after the header in the first block are reserved and should be zero. In this case the text and data sizes must both be multiples of 1024 bytes, and the pages of the file will be brought into the running image as needed, and not pre-loaded as with the other formats. This is especially suitable for very large programs and is the default format produced by ld(1).

The stack will occupy the highest possible locations in the core image: growing downwards from 0x7ffffff000. The stack is automatically extended as required. The data segment is only extended as requested by break(2).

After the header in the file follow the text, data, text relocation data relocation, symbol table and string table in that order. The text begins at the byte 1024 in the file for ZMAGIC format or just after the header for the other formats. The N_TXTTOFF macro returns this absolute file position when given the name of an exec structure as argument. The data segment is contiguous with the text and immediately followed by the text relocation and then the data relocation information. The symbol table follows all this; its position is computed by the N_SYMOFF macro. Finally, the string table immediately follows the symbol table at a position which can be gotten easily using N_STROFF. The first 4 bytes of the string table are not used for string storage, but rather contain the size of the string table; this size INCLUDES the 4 bytes, the minimum string table size is thus 4.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the
include file as follows:

/*
 * Format of a symbol table entry.
 */
struct nlist {
    union {
        char *n_name; /* for use when in-core */
        long n_strx; /* index into file string table */
    } n_un;
    unsigned char n_type; /* type flag, i.e. N_TEXT etc; see
    char n_other;
    short n_desc; /* see <stab.h> */
    unsigned n_value; /* value of this symbol (or sdb of
} n_type
#endif

#define N_UNDF 0x0  /* undefined */
#define N_ABS 0x2  /* absolute */
#define N_TEXT 0x4  /* text */
#define N_DATA 0x6  /* data */
#define N_BSS 0x8  /* bss */
#define N_COMM 0x12 /* common (internal to ld) */
#define N_FN 0x1f /* file name symbol */
#define N_EXT 0x1 /* external bit, or'ed in */
#define N_TYPE 0x1e /* mask for all the type bits */
#endif

/*
 * Other permanent symbol table entries have some of the N_STAB bit
 * These are given in <stab.h>
 */
#define N_STAB 0xe0 /* if any of these bits set, don't
*/

/*
 * Format for namelist values.
 */
#define N_FORMAT "%.8x"

In the a.out file a symbol's n_un.n_strx field gives an
index into the string table. A n_strx value of 0 indicates
that no name is associated with a particular symbol table
entry. The field n_un.n_name can be used to refer to the
symbol name only if the program sets this up using n_strx
and appropriate data from the string table.

If a symbol's type is undefined external, and the value
field is non-zero, the symbol is interpreted by the loader.
ld as the name of a common region whose size is indicated by the value of the symbol.

The value of a byte in the text or data which is not a portion of a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a byte in the text or data involves a reference to an undefined external symbol, as indicated by the relocation information, then the value stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the bytes in the file.

If relocation information is present, it amounts to eight bytes per relocatable datum as in the following structure:

```c
/*
 * Format of a relocation datum.
 */

struct relocation_info {
    int r_address;        /* address which is relocated */
    unsigned r_symbolnum:24, /* local symbol ordinal */
    r_pcrel:l,        /* was relocated pc relative already */
    r_length:2,        /* 0=byte, 1=word, 2=long */
    rExtern:1,        /* does not include value of sym rE */
    :4;                /* nothing, yet */
};
```

There is no relocation information if a_trsize+a_drsize==0. If rExtern is 0, then r_symbolnum is actually a n_type for the relocation (i.e. N_TEXT meaning relative to segment text origin.)

SEE ALSO
adb(1), as(1), ld(1), nm(1), sdb(1), stab(5), strip(1)

BUGS
Not having the size of the string table in the header is a loss, but expanding the header size would have meant stripped executable file incompatibility, and we couldn't hack this just now.
NAME
acct - execution accounting file

SYNOPSIS
#include <sys/acct.h>

DESCRIPTION
Acct(2) causes entries to be made into an accounting file for each process that terminates. The accounting file is a sequence of entries whose layout, as defined by the include file is:

#include <sys/acct.h>

typedef unsigned short comp_t; /* "floating pt": 3 bits base 8 exp, */
struct acct
{
    char    ac_comm[10];  /* Accounting command name */
    comp_t    ac_utime;  /* Accounting user time */
    comp_t    ac_stime;  /* Accounting system time */
    comp_t    ac_etime;  /* Accounting elapsed time */
    time_t    ac_btime;  /* Beginning time */
    short     ac_uid;    /* Accounting user ID */
    short     ac_gid;    /* Accounting group ID */
    short     ac_mem;    /* average memory usage */
    comp_t    ac_io;     /* number of disk IO blocks */
    dev_t     ac_tty;    /* control typewriter */
    char     ac_flag;    /* Accounting flag */
};

extern struct acct     acctbuf;
extern struct inode     *acctcp; /* inode of accounting file */

#define AFORK 01            /* has executed fork, but no exec */
#define ASU  02             /* used super-user privileges */

If the process does an exec(2), the first 10 characters of the filename appear in ac_comm. The accounting flag contains bits indicating whether exec(2) was ever accomplished, and whether the process ever had super-user privileges.
SEE ALSO
acct(2), sa(1)
NAME
aliases - aliases file for delivermail

SYNOPSIS
/usr/lib/aliases

DESCRIPTION
This file describes user id aliases that will be used by
/etc/delivermail. It is formatted as a series of lines of the form
  name:addr1,addr2,...addrn
The name is the name to alias, and the addr are the
addresses to send the message to. Lines beginning with
white space are continuation lines. Lines beginning with
' #' are comments.

Aliasing occurs only on local names. Loops can not occur,
since no message will be sent to any person more than once.

This is only the raw data file; the actual aliasing information is placed into a binary format in the files
/usr/lib/aliases.dir and /usr/lib/aliases.pag using the program newaliases(5). A newaliases command should be executed
each time the aliases file is changed for the change to take effect.

SEE ALSO
newaliases(l), dbm(3), delivermail(8)

BUGS
Because of restrictions in dbm(3) a single alias cannot con-
tain more than about 1000 bytes of information. You can get
longer aliases by `chaining'; i.e. make the last name in
the alias by a dummy name which is a continuation alias.
NAME
ar - archive (library) file format

SYNOPSIS
#include <ar.h>

DESCRIPTION
N.B.: This archive format is new to this distribution. See
old(8) and arcv(1) for programs to deal with the old format.

The archive command ar is used to combine several files into
one. Archives are used mainly as libraries to be searched
by the link-editor ld.

A file produced by ar has a magic string at the start, fol­
lowed by the constituent files, each preceded by a file
header. The magic number and header layout as described in
the include file are:

#define ARMAG "!<arch>
#define SARMAG 8
#define ARFMAG "\n"

struct ar_hdr {
    char ar_name[16];
    char ar_date[12];
    char ar_uid[6];
    char ar_gid[6];
    char ar_mode[8];
    char ar_size[8];
    char ar_fmag[2];
};

The name is a blank-padded string. The ar_fmag field con­
tains ARFMAG to help verify the presence of a header. The
other fields are left-adjusted, blank-padded numbers. They
are decimal except for ar_mode, which is octal. The date is
the modification date of the file at the time of its inser­
tion into the archive.

Each file begins on a even (0 mod 2) boundary; a new-line is
inserted between files if necessary. Nevertheless the size
given reflects the actual size of the file exclusive of
There is no provision for empty areas in an archive file.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

SEE ALSO
ar(1), ld(1), nm(1)

BUGS
File names lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.
NAME
core - format of memory image file

DESCRIPTION
UNIX writes out a memory image of a terminated process when any of various errors occur. See signal(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The memory image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The maximum size of a core file is limited by vlimit(2). Files which would be larger than the limit are not created.

The core file consists of the u. area, which currently consists of 6 pages, beginning with a user structure as given in /usr/include/sys/user.h. The kernel stack grows from the end of this 6 page region. The remainder of the core file consists first of the data pages and then the stack pages of the process image.

In general the debugger adb(1) is sufficient to deal with core images.

SEE ALSO
adb(1), signal(2), vlimit(2)
NAME
dir - format of directories

SYNOPSIS
   #include <sys/types.h>
   #include <sys/dir.h>

DESCRIPTION
   A directory behaves exactly like an ordinary file, save that
   no user may write into a directory. The fact that a file is
   a directory is indicated by a bit in the flag word of its
   i-node entry; see filsys(5). The structure of a directory
   entry as given in the include file is:

   #ifndef DIRSIZ
   #define DIRSIZ 14
   #endif
   struct direct
   {
      ino_t   d_ino;
      char    d_name[DIRSIZ];
   };

   By convention, the first two entries in each directory are
   for '.' and '..'. The first is an entry for the directory
   itself. The second is for the parent directory. The mean­
   ing of '..' is modified for the root directory of the master
   file system " ." " /" " " " where '..' has the same meaning as '.'.

SEE ALSO
   filsys(5)
NAME

dump, ddate - incremental dump format

SYNOPSIS

#include <sys/types.h>
#include <sys/inode.h>
#include <dumprestor.h>

DESCRIPTION

Tapes used by dump and restor(1) contain:

- a header record
- two groups of bit map records
- a group of records describing directories
- a group of records describing files

The format of the header record and of the first record of each description as given in the include file <dumprestor.h> is:

#define NTREC 10
#define MLEN 16
#define MSIZ 4096

#define TS_TAPE 1
#define TS_INODE 2
#define TS_BITS 3
#define TS_ADDR 4
#define TS_END 5
#define TS_CLRI 6
#define MAGIC (int) 60011
#define CHECKSUM (int) 84446

struct spcl {
    int c_type;
    time_t c_date;
    time_t c_ddate;
    int c_volume;
    daddr_t c_tapea;
    ino_t c_inumber;
    int c_magic;
    int c_count;
    struct dinode c_dinode;
    struct dates {
        char id_name[16];
        char id_incno;
        time_t id_ddate;
    } spcl;
} spcl;
The NTREC is the number of 1024 byte records in a physical tape block. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS entries are used in the c_type field to indicate what sort of header this is. The types and their meanings are as follows:

| TS_TAPE   | Tape volume label  |
| TS_INODE  | A file or directory follows. The c_dinode field is a copy of the disk inode and contains bits telling what sort of file this is. |
| TS_BITS   | A bit map follows. This bit map has a one bit for each inode that was dumped. |
| TS_ADDR   | A subrecord of a file description. See c_addr below. |
| TS_END    | End of tape record. |
| TS_CLRI   | A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped. |
| MAGIC     | All header records have this number in c_magic. |
| CHECKSUM  | Header records checksum to this value. |

The fields of the header structure are as follows:

c_type   The type of the header.
c_date   The date the dump was taken.
c_ddate  The date the file system was dumped from.
c_volume The current volume number of the dump.
c_tapea  The current number of this (1024-byte) record.
c_inumber The number of the inode being dumped if this is of type TS_INODE.
c_magic  This contains the value MAGIC above, truncated as needed.
c_checksum This contains whatever value is needed to make the record sum to CHECKSUM.
c_dinode This is a copy of the inode as it appears on the file system; see filsys(5).
c_count  The count of characters in c_addr.
c_addr   An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was
dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, TS_ADDR records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END record and then the tapemark.

The structure idates describes an entry of the file /etc/ddate where dump history is kept. The fields of the structure are:

id_name The dumped filesystem is '/dev/id nam'.
id_incno The level number of the dump tape; see dump(1).
id_ddate The date of the incremental dump in system format see types(5).

FILES
/etc/ddate

SEE ALSO
dump(8), dumpdir(8), restor(8), filsys(5), types(5)
NAME
environ - user environment

SYNOPSIS
extern char **environ;

DESCRIPTION
An array of strings called the 'environment' is made available by exec(2) when a process begins. By convention these strings have the form 'name=value'. The following names are used by various commands:

PATH  The sequence of directory prefixes that sh, time, nice(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by ':'. Login(1) sets PATH=:/usr/ucb:/bin:/usr/bin.

HOME  A user's login directory, set by login(1) from the password file passwd(5).

TERM  The kind of terminal for which output is to be prepared. This information is used by commands, such as nroff or plot(1), which may exploit special terminal capabilities. See /etc/termcap (termcap(5)) for a list of terminal types.

SHELL  The file name of the users login shell.

TERMCP A string describing the terminal in TERM, or the name of the termcap file, see termcap(5), termlib(3).

EXINIT  A startup list of commands read by ex(1), edit(1), and vi(1).

USER  The login name of the user.

Further names may be placed in the environment by the export command and 'name=value' arguments in sh(1), or by the setenv command if you use csh(1). Arguments may also be placed in the environment at the point of an exec(2). It is unwise to conflict with certain sh(1) variables that are frequently exported by `./profile' files: MAIL, PS1, PS2, IFS.

SEE ALSO
csh(1), ex(1), login(1), sh(1), exec(2), system(3), term-lib(3), termcap(5), term(7)
NAME
group - group file

DESCRIPTION
Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES
/etc/group

SEE ALSO
newgrp(1), crypt(3), passwd(1), passwd(5)

BUGS
The passwd(1) command won't change the passwords.
NAME
 mtab - mounted file system table

DESCRIPTION
 Mtab resides in directory /etc and contains a table of devices mounted by the mount command. Umount removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last '/' is thrown away.

This table is present only so people can look at it. It does not matter to mount if there are duplicated entries nor to umount if a name cannot be found.

FILES
 /etc/mtab

SEE ALSO
 mount(8)
NAME

passwd - password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
numerical group ID
user's real name, office, extension, home phone.
initial working directory
program to use as Shell

The name may contain `&', meaning insert the login name.
This information is set by the chfn(1) command and used by
the finger(1) command.

This is an ASCII file. Each field within each user's entry
is separated from the next by a colon. Each user is
separated from the next by a new-line. If the password
field is null, no password is demanded; if the Shell field
is null, then /bin/sh is used.

This file resides in directory /etc. Because of the
encrypted passwords, it can and does have general read per­
mission and can be used, for example, to map numerical user
ID's to names.

Appropriate precautions must be taken to lock the file
against changes if it is to be edited with a text editor;
vipw(8) does the necessary locking.

FILES

/etc/passwd

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(5),
chfn(1), finger(1), vipw(8), adduser(8)

BUGS

A binary indexed file format should be available for fast
access.

User information (name, office, etc.) should be stored else­
where.
NAME

ttys - terminal initialization data

DESCRIPTION

The ttys file is read by the init program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is used as an argument to getty(8), which performs such tasks as baud-rate recognition, reading the login name, and calling login. For normal lines, the character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (Getty will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, /dev.

FILES

/etc/ttys

SEE ALSO

init(8), getty(8), login(1)
NAME
  ttytype - data base of terminal types by port

SYNOPSIS
  /etc/ttytype

DESCRIPTION
  Ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to it. There is one line per port, containing the terminal kind (as a name listed in termcap (5)), a space, and the name of the tty, minus /dev/.

  This information is read by tset(1) and by login(1) to initialize the TERM variable at login time.

SEE ALSO
  tset(1), login(1)

BUGS
  Some lines are merely known as "dialup" or "plugboard".
NAME
types - primitive system data types

SYNOPSIS
#include <sys/types.h>

DESCRIPTION
The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

/* a la types.h 4.1 81/03/21 */

/*
 * Basic system types and major/minor device constructing/busting macro
 */

/* major part of a device */
define major(x) ((int)((unsigned)(x)>>8)&0377)

/* minor part of a device */
define minor(x) ((int)((x)&0377))

/* make a device number */
define makedev(x,y) ((dev_t)(((x)<<8) | (y)))

typedef unsigned char u_char;
typedef unsigned short u_short;
typedef unsigned int u_int;
typedef unsigned long u_long;

/* SHOULD USE long RATHER THAN int HERE BUT IT WOULD GIVE LINT ON THE K
 /* GASTRIC DISTRESS AND DON'T HAVE TIME TO FIX THAT JUST NOW */
typedef struct __physadr { int r[l]; } *physadr;
typedef int daddr_t;
typedef char * caddr_t;
typedef u_short ino_t;
typedef int time_t;
typedef int label_t[l3]; /* regs d2-d7, a2-a7, pc */
typedef short dev_t;
typedef int off_t;
typedef int mem_t;
typedef u_long tim_id_t; /* timeout id */
typedef int (*faddr_t)(); /* Pointer to a function */
#ifdef KERNEL
typedef int vector_t; /* interrupt vectors */
#define NULLVECTOR ((vector_t) -1)
#endif KERNEL

typedef u_char bool_t;
#ifdef YES
#define YES 1
#define NO 0
#endif YES

#define MAX_LONG 0x7FFFFFFF
#define MAX_INT 0x7FFFFFF
#define MAX_SHORT 0x7FF
#define MAX_CHAR 0x7F
#define MAX_U_LONG 0xFFFFFFFF
#define MAX_U_INT 0xFFFFFFFF
#define MAX_U_SHORT 0xFFFF
#define MAX_U_CHAR 0xFF
#endif lint
#define void int /* so berkeley void coersions will work */
#endif

The form daddr_t is used for disk addresses except in an i-node on disk, see filesys(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The label_t variables are used to save the processor state while another process is running.

SEE ALSO
filesys(5), time(2), lseek(2), adb(1)
NAME
utmp, wtmp — login records

SYNOPSIS
#include <utmp.h>

DESCRIPTION
The utmp file allows one to discover information about who is currently using UNIX. The file is a sequence of entries with the following structure declared in the include file:

```c
struct utmp {
    char ut_line[8];      /* tty name */
    char ut_name[8];      /* user id */
    long ut_time;         /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of time(2).

The wtmp file records all logins and logouts. Its format is exactly like utmp except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name '"' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names '|' and '}' indicate the system-maintained time just before and just after a date command has changed the system's idea of the time.

Wtmp is maintained by login(1) and init(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by ac(8).

FILES
/etc/utmp
/usr/adm/wtmp

SEE ALSO
login(1), init(8), who(1), ac(8)
NAME
uuencode - format of an encoded uuencode file

DESCRIPTION
Files output by uuencode(1) consist of a header line, followed by a number of body lines, and a trailer line. Uudecode(1) will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters "begin". The word begin is followed by a mode (in octal), and a string which names the remote file. A space separates the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of "end" on a line by itself.

SEE ALSO
uuencode(1), uudecode(1), uusend(1), uucp(1), mail(1)
NAME
wtmp - user login history

DESCRIPTION
This file records all logins and logouts. Its format is exactly like utmp(5) except that a null user name indicates a logout on the associated typewriter. Furthermore, the typewriter name `~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with typewriter names `|' and `}' indicate the system-maintained time just before and just after a date command has changed the system's idea of the time.

Wtmp is maintained by login(l) and init(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by ac(l).

FILES
/usr/adm/wtmp

SEE ALSO
utmp(5), login(l), init(8), ac(l), who(l)