LANGUAGE DEVELOPMENT TOOLS

DEVELOPMENT TOOLS
FOR FORTUNE LANGUAGES

FORTUNE SYSTEMS
300 Harbor Boulevard
Belmont, CA 94002

Language Development Tools Guide

FORTUNE SYSTEMS
300 Harbor Boulevard
Belmont, CA 94002

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

Ordering Language Development Tools

Order No.: 1002146-01 February 1984 for manual with disk
1002145-01 February 1984 for manual without disk

Consult an authorized Fortune Systems Representative for copies of manuals and technical 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

Fortune Systems Corporation offers three systems languages for development purposes: C, FORTRAN, and Pascal. The Language Development Tools is a language-independent companion document to the guides that accompany the compiler disks for these languages. It is designed to meet the needs of experienced programmers for programming in the FOR:PRO environment. (FOR:PRO is an enhanced version of Bell Laboratories' Version 7 UNIX operating system, modified to run on Fortune computers.)

Language Development Tools describes the software tools provided on the Language Development Tools set of disks with which the guide is shipped. These tools aid in the efficient management of programs and files in any or all of the three systems languages.

ORGANIZATION OF THIS BOOK

Language Development Tools contains five chapters. Chapters 1 and 2 address the needs of both experienced and novice programmers. Chapters 3 through 5 are intended for experienced system developers. Each chapter is summarized below.

Language Development Tools is not intended as a language reference or tutorial. If you need instructions on a particular language, refer to the manuals that accompany the compiler disk for the language.

- **Chapter 1** contains procedures for installing and backing up the Language Development Tools and language compiler disks.

- **Chapter 2** explains how to create a program using **ed**, the FOR:PRO line editor.

- **Chapter 3** describes the FOR:PRO tools **archive, ranlib, strip,** and **make.** With these tools you can create and update library files; convert these files into random libraries for faster loading; remove the symbol table attached to a program for debugging purposes; and make up-to-date versions of programs consisting of many files.

- **Chapter 4** documents the **size, name,** and **ctags** tools, which aid you in determining the size and names of object files and the location of functions in source files.

- **Chapter 5** provides a detailed explanation of the Fortune Symbolic Debugger (**fdb**). It includes **fdb** commands and special rules, instructions for debugger use, and debugger messages.

Contents

THE LANGUAGE DEVELOPMENT TOOLS: AN OVERVIEW

This section presents an overview of the Language Development Tools and the three system languages. Conventions used in this document are summarized to prepare you for using the rest of this guide.

THE TOOLS

The Language Development Tools are a set of commands and supporting files for use in compiling, executing, and debugging programs in the FOR:PRO environment.

The Single-User FOR:PRO set is shipped with every Fortune computer and contains all of the features necessary to start using the Fortune system.

The Development Utilities set extends the capabilities of FOR:PRO with additional features that are useful for programmers. The Development Utilities supplements the Language Development Tools and is available as a separate software package.

The Language Development Tools set, which this guide addresses, is for systems and applications programmers. The software that is provided with this binder can assist you in the development of programs. These tools are designed for managing programs and files more efficiently, for examining source and object code, and for debugging programs. The set of tools include the following items.

For managing files and programs

make Produces final versions of programs that are composed of many separate files and files that require many complex commands in their production.

archive Creates and updates libraries of object files used by the linking loader.

ranlib Converts archives to a random library for faster loading by the linking loader.

strip Saves space by removing the symbol table and the relocation bits that are attached to the output file by the assembler and loader.

For examining source and object code

size Displays the size of an object file in bytes.

name Prints the name list (symbol table) of each object file in the argument list.

ctags Creates a "tags" file that gives the line number locations of specified functions in C, FORTRAN, and Pascal source programs.

For debugging programs

fdb The Fortune symbolic debugger is a high-level debugging tool designed to be language-independent. It can therefore serve as a debugger for the compiled high-level languages supported on the Fortune system.

THE LANGUAGES

The three languages available on the Fortune system are: C, FORTRAN, and Pascal.

C is a general purpose programming language that is perhaps best described as a "powerful assembly language." It offers the programmer the advantages of coding brevity, a wide variety of data structures, modern flow-control constructs, fast floating-point, single and double processor, and operators. C is well suited to system software development (most of UNIX as well as Fortune's FOR:PRO are written in C) and has been used successfully in a wide range of commercial, scientific, and data base applications.

FORTRAN is a high level, problem oriented programming language that allows the programmer to communicate with the Fortune computer in a semi-English scientific language. When using FORTRAN (an acronym for Formula Translator), the programmer defines a problem through mathematical relationships and formulas. (Ratfor and EFL programs can also be run on a Fortune system with an f77 compiler. Fortune offers the processors for these FORTRAN dialects, but does not support them. Documentation must be obtained from Bell Laboratories' UNIX programmer's manuals. Ratfor is documented in Volume 2b of that set; EFL is in volume 2c.)

Pascal is a general purpose, block-structured programming language that promotes the writing of well-structured, readable programs. Pascal encourages and supports advanced program design approaches such as top-down program and data structure design, structured coding, program modularity, stepwise refinement, and team work on programming.

Each of these languages is a separate software and documentation package. However, one set of Language Development Tools supports all three languages. The guides to each language are thus designed to be inserted into this binder. See Figure i-1 for an overview of the software and documentation available in the entire Language Development Tools set.

CONVENTIONS USED IN THIS GUIDE

This guide and the three language guides are written for programmers. These books do not attempt to teach the basics of programming. A list of reference texts is provided with each guide for new users of any of the three languages.

Throughout the manual, you will find examples and descriptions of syntax for commands and other elements of the languages. This guide assumes you are using the standard Bourne shell, hence the **$** prompt, indicating the start of a command line. In these examples, the following conventions are used:

- Commands to FOR:PRO are almost always entirely lowercase letters, as is the case with most UNIX commands. Occasionally some of the command options may be uppercase. Be careful to type the commands exactly as shown in the text.

- In syntax statements, any **input** that you must type is shown in **boldface.** Examples and filenames are always in boldface.

- Words that you must replace with your own text are <u>underlined</u>. Such items are also referred to as command-line parameters.

- Brackets [] indicate one or more options that you may or may not want to use.

- The hyphen preceding an option, as in the command

  ls -l

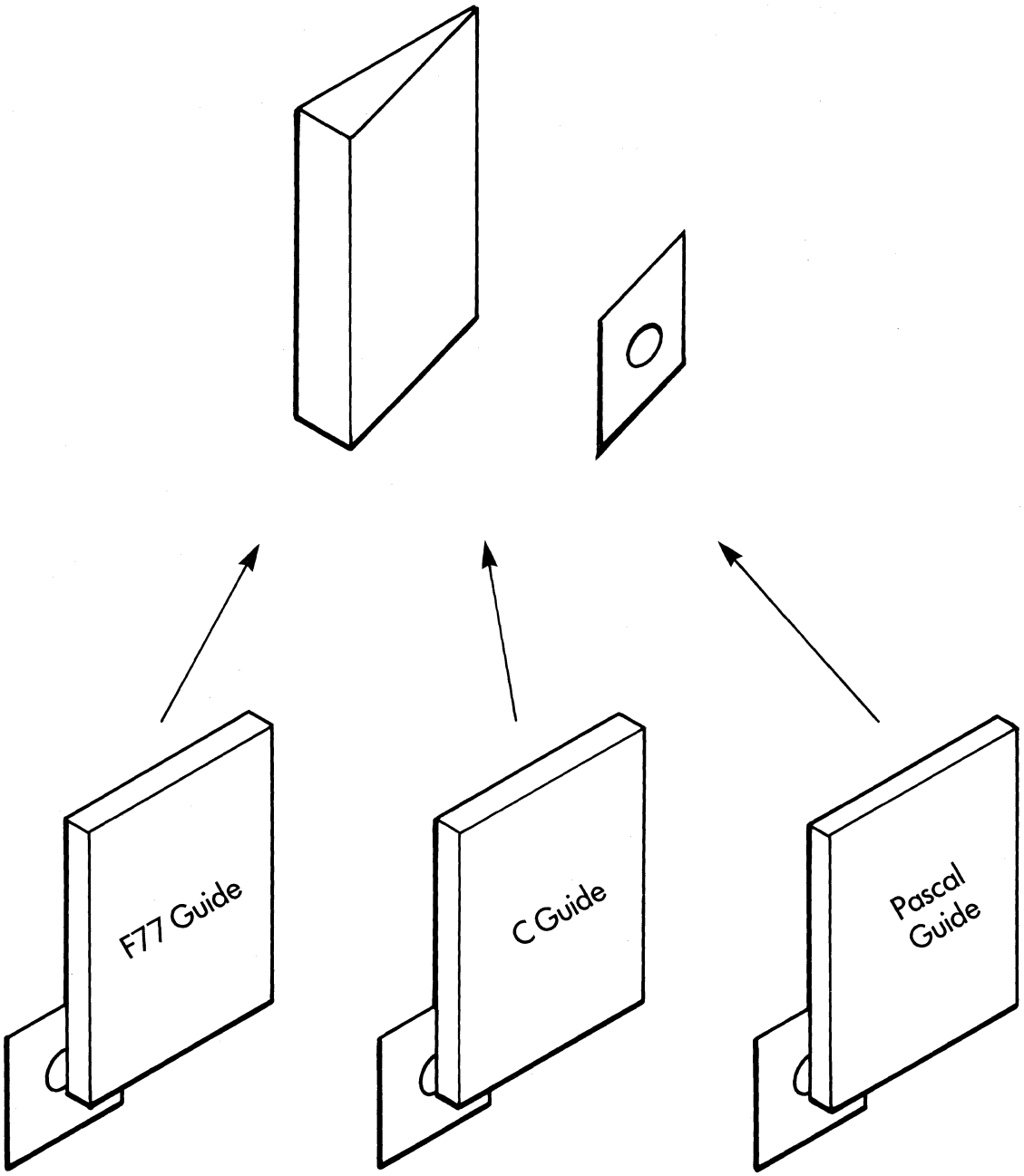
  must always be typed. Hyphens on a command line signal an option.

- Braces {} indicate that at least one of the enclosed options must be included in the command.

- Ellipses ... mean that the preceding option may be repeated.

- Commands are ended and executed by pressing the **Return** key.

The Language Development Tools set includes this binder, text, and software.



Each language comes with a language guide. Insert your guide (or guides) in this binder.

Figure i-1. Language Development Tools Set

1 INSTALLING THE LANGUAGE DEVELOPMENT TOOLS

To install the Language Development Tools, follow the procedures in this chapter.

SETTING THE SYSTEM CONFIGURATION

To install the Language Development Tools and/or the language compilers, you may need to reconfigure your Fortune system.

If your system was configured to support the Language Development Tools or compilers during the installation of the FOR:PRO 1.7 operating system, it will not be necessary to make any changes.

On the Configuration menu, the "Max Process Size" must be set to at least 256K and the "Appx # Users" must be set to the number of terminals on your system. You should also review the number of swap units for which your system is configured.

Information on setting the system configuration is provided in the FOR:PRO Installation Instructions. Refer to Chapter 1 on "Determining SWAP Space Requirements" and Chapter 6, "Changing the Configuration Menu."

INSTALLING THE PROGRAMS AND FILES ON THE HARD DISK

Before you use any of the system programming languages (C, FORTRAN, or Pascal), you must install the programs and files from the master flexible disks to the hard disk on the system you are using. You need a minimum 512K RAM memory configuration to use the languages.

To install the programs and files, log in as manager, and use the menu system as follows:

1. On the Global menu, select **S5 Product Maintenance** and press the **Return** key.

2. Enter **i** for the "Install" selection and press the **Return** key.

3. Insert the Language Development Tools flexible disk in the flexible disk drive device and press **Return.**

4. Enter **y** when asked if you want to proceed with installation and whether the max process size is 256K.

5. Remove the Language Development Tools flexible disk when you receive the message that the installation is successfully completed. An error message will appear if the installation is not successful.

6. Repeat this process with the language compiler disk(s).

FORMATTING AND COPYING DISKS

Make a copy of your software as soon as possible after it has been installed on the hard disk. Use the procedure below to format one or more blank flexible disks, and then copy your software onto it (them) from your master disk. To perform this procedure you must be logged in as manager.

1. On the Global menu, select **S1 System Utilities**, and press the **Return** key.

2. Next, select **32 Format Flexible Disk** on the System Utilities menu.

3. Insert the flexible disk that is to be formatted.

4. Enter **y** to the question, "Do you want to continue?"

5. When you receive the completion message, either remove the flexible disk or leave it in the drive if you are copying it as the next operation.

BACKING UP MASTER DISKS

This procedure is used to back up your master disks.

1. On the Global menu, select **S5 Product Maintenance** and press the **Return** key.

2. On this new menu, select **b** for back up.

3. Enter:

   ds for the development set
   cc for the C compiler
   f77 for the FORTRAN compiler
   pc for the Pascal compiler

4. Answer **y** to the question, "Do you want to continue?"

5. Insert a formatted disk and press the **Return** key.

6. When you receive the completion message, remove the disk.

7. Repeat the process to back up each compiler.

2 ENTERING AND EDITING PROGRAMS

Two basic types of text editors are available on the FOR:PRO operating system: line editors and screen editors. Line editors allow you to create text and then edit it a line at a time. With screen editors you can edit text a screenful at a time, taking advantage of the characteristics of the video screen.

The screen editor, **vi**, is provided with the Development Utilities set. The line editor, **ed**, is the text editor available on the FOR:PRO Single-User set. It's easy to learn and adequate for creating simple ASCII text files.

This chapter describes the basic steps in using **ed** and the commands associated with it.

To do extensive text formatting on a file created with **ed**, use the commands associated with the text formatting program **nroff**. This program is available with the Development Utilities Product. Instructions for its use are included in the FOR:PRO Programmer's Manual. For more information on the text editor **ed**, see, Introduction to FOR:PRO, Chapter 1, "The Shell Environment."

ed: INVOKING THE EDITOR

To use **ed**, you should be at the command level. From there, invoke **ed** like any other FOR:PRO command, using the syntax:

```
$ ed filename
```

Use this syntax whether you are creating a new file or editing an existing one.

CREATING A NEW FILE WITH ed

The **ed** editor has two modes of operation: the **append** mode creates text, and the **edit** mode modifies text. To create a new file, first type **ed** and the name of the file as shown above. The system responds to this command with:

```
?filename
```

This response indicates that this is a new file. The **?** prompt indicates that you are in the edit mode.

To enter the append mode, type an **a** and press the **Return** key. Once you are in append mode, you simply type the text you want to include in the file.

If you make a mistake while typing, you can correct the mistake, provided it is on the current line. Use the **Backspace** key to move the cursor over the mistake; this action erases the text. You can then retype the text as you want it.

To leave the append mode, type a period at the beginning of a new line and then press the **Return** key.

At this point, you can either edit the file or save it for modification at a later time. If you don't plan to edit the file immediately after creating it, save its contents by issuing a **w** command. This command writes the current version of the file onto the hard disk. Type a **q** to leave **ed** and return to the shell.

You should be aware of the following rules when creating files:

First, filenames must not be longer than 14 characters and must not begin with the following special characters.

, $ > < ? * & ; .

Second, FOR:PRO, like all UNIX systems, distinguishes between uppercase and lowercase letters, so type filenames carefully.

Last, be aware that the cursor control keys (grey keys with arrows) do not work in **ed**.

EDITING TEXT USING ed

To modify existing files, you can use some of the **ed** single-character commands to change text on a line-by-line basis. The **ed** editor offers a full range of commands; those that you will use most often are described in Table 2-1. (For a complete list, see the FOR:PRO Programmer's Manual.)

Table 2-1. Table of ed Commands

Command	Description
a	Appends text after the current line. This is the same command that is used when creating a file. The syntax of the **a** command is (current line) **a <return>** **(New text--as many lines as needed)** **. <return>**
d	Deletes the text in the indicated line or lines. The syntax of **d** is **[line number]d** or **[line number],[line number]d**
i	Inserts text above the current line. You can insert as much text as necessary, indicating the end of text by typing a period on the line below the last line of text, followed by a RETURN. The syntax of **i** is **i <return>** **(New text--as many lines as needed)** **. <return>**
p	Prints on the screen the line or range of lines specified. The syntax of **p** is **[line number]p** **[line number],[line number]p** Use the **$** symbol to specify the last line of the file. You can also print the contents of a line by typing that line number, such as **3**.
q	Quits the **ed** editor and returns to the shell.
s	Replaces a new string of text for an old string in the given line, in a range of lines, or throughout the entire file.

Command	Description
	Use the following format to change the current line of text. The **g** option indicates that you want to make the indicated change to every occurrence of **old.text** on that line. **s/old.text/new.text/[g]** Use this format to change text over a range of lines indicated by x,y: x,y **s/old.text/new.text/** The format below adds the new text string to the beginning of the current line: **s/^/newtext/** This format adds the new text string to the end of the current line: **/s/$/newtext**
u	"Undoes" the results of the last editing command issued at the current line. It applies only to editing commands that may have modified the line. The contents of the line are restored to their state before the last command was issued.
w	Writes the contents of the file to the hard disk.
/text/	Searches for the characters between the slashes. (If you need to use slashes in the text, type one backslash before the slash, as in: \/ .)
$	Moves you to the last line of a file and prints its contents.
+[nn]	Moves you from the current line to the next line; with the **nn** option, this command moves you forward nn lines.
-[nn]	Moves you from the current line to the preceding line; the **nn** option moves you backwards nn lines towards the top of the file.
nn	Moves you to the line number **nn** and prints that line.

A SAMPLE ed SESSION

If you do not have any files to experiment with, you can create the following sample file to try out **ed**.

Begin the process by typing the following:

```
$ ed file.p
```

The system responds:

```
?file.p
```

Enter the append mode by typing **a** and pressing the **Return** key. You won't get any further system responses until you leave the append mode.

Type the following short Pascal program:

```
program hello(output);
begin
writeln('Hello Universe')
end.
.
```

The single period on the last line of the file represents the end of text. Now type **1,$ p** to display the entire file.

Type **w** to write the text you just typed onto the hard disk. The system responds with a number, representing the number of characters (bytes) in the file.

Type **q** to leave **ed** and return to the shell. You should see the **$** prompt on your screen.

3 MANAGING FILES AND PROGRAMS

Several LDT tools allow you to manipulate, create, and maintain files and programs. With the archive (**ar**) command, for example, you can place any number of modules (usually object) into a single archive, or library file. Another tool, **ranlib**, works on these files to allow faster searching by the loader (**ld**). The **strip** command removes symbol and relocation information from an executable module to make the executable smaller and to save space. The final tool described in this chapter, **make**, provides a way to easily manage and maintain large programming projects and produce files which may involve complex or repetitive processing. These tools are used as follows.

archive (ar): CREATING UP-TO-DATE LIBRARY FILES

The archive (**ar**) tool is used primarily to create and update library files searched by the loader. Groups of files are maintained in one archive file. This version of archive (**ar**) uses an ASCII format archive header that can be shared among various machines running UNIX.

The syntax for **archive** is

    $ **ar** key [posnamell] afile name...

In this command line, **posname** is the file name you use to indicate position (this optional file name is unnecessary if you use the **ranlib** tool); **afile** is the name you assign to the archive file; and **name(s)** is the file (or files) that is in or to be added to the archive file.

The **key** is a set of characters that denote an instruction for manipulating the files. There are seven possible instruction characters. These may be used with one or more options. Table 3-1 lists the instructions; Table 3-2 lists the options.

The following are examples of the use of the **archive** tool:

    $ ar rv devices.a printer.o tty.o

This example creates an archive library called **devices.a**. It consists of two object modules, **print.o** and **tty.o**.

To extract an object file from an archive, type

    $ ar xv devices.a tty.o

Table 3-1. Keys for Use in the archive Command Line.

Key Character	Description of Instruction
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 that have been modified since one of the archive files were created are replaced. If you use an optional positioning character a, b or i, the **posname** argument must be included. It specifies that new files are to be positioned following (a) or before (b or i) the position named. Otherwise, new files are placed at the end.
q	Quickly appends the named files to the end of the archive file, disregarding any optional positioning characters and without checking to determine if the added files are already in the archive.
t	Prints a table of contents for the archive file. If no names are printed, all the files in the archive are included in the table. If names are printed, only those files with names are included in the table.
p	Prints the contents of named files in the archive.
m	Moves the named files to the end of the archive. If you include a positioning character, then the **$posname** argument must be used as with r to specify where the files are to be moved.
x	Extracts the named files and places them in the working directory. If you give no names, all files in the archive are extracted; x does not, however, alter the archive file.

This command extracts the **file tty.o** from the archive, and places it unchanged into the current directory.

To delete an object file, type

```
$ ar  d  devices.a  tty.o
```

The file **tty.o** is deleted from the archive.

Table 3-2. Options for Use in the **ar** Command Line.

Option	Description
-v	This is the "verbose" option. With it, 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. The **p** key prints the filename before each file.
-c	The create option suppresses the usual message produced when a file is created.
-l	The local option places files in the local directory rather than in **/tmp**, where temporary files are normally placed.
-u	The update option replaces the contents of an old archive with new contents.
-a	This option is used in conjunction with the r instruction to place new files after a specified position.
-b or -i	This option is used in conjunction with the instruction to place new files before a specified position.

ranlib: LOADING ARCHIVE FILES MORE RAPIDLY

The **ranlib** tool converts each archive to a random library that can be searched. As described above, an archive is a file that contains a collection of **.o** object files.

When invoked, **ranlib** adds a table of contents named **__.SYMDEF** to the beginning of the archive. This entry is then searched by the loader.

The syntax for **ranlib** is

```
$ ranlib archive
```

where **archive** is the name of the archive file containing a collection of **.o** object files.

strip: REDUCING THE SIZE OF A DEBUGGED PROGRAM

The **strip** tool removes the symbol table and the relocation bits which are attached to the output file produced by the assembler and loader. You can use this tool to save space after you have debugged a program.

The effect of **strip** is the same as that of the **-s** option of the **ld** command. It reduces the size of a file.

The syntax for **strip** is

```
$ strip filename
```

Sample Program for Use with the strip Tool

You may use this simple program to see the effect of strip.

```
/* complete the squares of the numbers 0 to 9 */
main()
}
     register int x,i;
     for (i=0; i < 10; i++){
          x = i*i;
          printf("i = %d i squared = %d", i,x);
          {

{
```

To see the effect of **strip**, enter the following:

```
$ cc strip.c          Produces a.out

$ ls -l  a.out        long listing of a.out, before
                      strip is invoked, produces

-rwxrwxr-x 1 user     8739 Dec30 14:00 a.out

$ strip a.out         produces the following
$ ls -l a.out

-rwxrwxr-x1, user     6892 Dec 30 14:00 a.out
                      the actual size of the
                      stripped a.out file depends
                      on the version of the language
                      compiler and the version of LDT
                      that you are using
```

make: CREATING UP-TO-DATE VERSIONS OF PROGRAMS

make is a sophisticated tool that is most commonly used to produce final, up-to-date versions of the following:

- Programs that are composed of many separate files
- Files with production requirements that require the typing of several complex commands.

make works by reading a file (a **makefile**) into which you have placed FOR:PRO commands. These commands are in groups, and each group has a header or title. The left-hand part of the title is either the name of a file which some of the commands beneath it produce, or is a symbolic name used for reference within the makefile. If it is a symbolic name, the commands beneath it do not produce it; but are associated with it. The file or symbolic name serves as a title for the group of commands directly beneath it. This part of the title is called the target.

In addition to the target, many titles in a makefile list dependency files. The basic format of the title-command construction is

```
target: dependency files   This line is the title.
<tab>                      command
<tab>                      command2
```

A dependency file is not always a file: Like a target, it can be a symbolic name used within the makefile. Dependency file lists tell **make** that before it executes the commands associated with the current target, it must execute the commands associated with the dependency files. Dependency files are usually listed as targets, or titles for another group of commands elsewhere in the makefile. For example:

```
a:b x      a depends on b and x.
           commands associated with a

b:x        b depends on x.
           commands associated with b

x:         x has no dependency files.
           commands associated with x
```

With this makefile, **make** would first execute the commands listed under x, then **b**, then **a**.

The main purpose of **make** is usually to perform some sort of compilation, via a **lex**, an **f77**, or an **nroff** processor, because avoiding unnecessary compilations requires knowledge of when a file was last edited. **make** avoids repetitive processing by checking the time and date when a file was last edited, and

compares that time to the time when the file was last processed. It does not reprocess a file unless the file or any of its dependency files has been edited since the last time it was processed.

If a target is a symbolic name, rather than the name of an actual file in a directory, **make** cannot know when it last executed the associated commands. In this case, **make** re-executes all the commands.

How **make** Works

In your directory you have three files that constitute one program. The files are named **a.c**, **b.c**, and **c.c**. To compile, link, and execute these files, you normally use these commands:

```
$ cc -c a.c b.c c.c   Resulting in a.o, b.o, and c.o.

$ cc a.o b.o c.o      Linking the .o into a.out.

$ a.out               Executing a.out.
```

Each time you edit one of the files, you need to compile that file again, then re-link all three files and execute the resulting file. Your task could be simplified if you typed only one command:

```
$ make
```

To use **make**, however, you need to produce a makefile. The makefile is produced through the editor. For now, consider its correct name to be **makefile** or **Makefile**. The makefile that would execute the above commands is

```
a.out: a.o b.o c.o
       cc a.o b.o c.o
       a.out
a.o : a.c
       cc -c a.c
b.o : b.c
       cc -c b.c
c.o : c.c
       cc -c c.c
```

(This makefile could be simpilified; the above example is for illustrative purposes only. Ensuing sections of this chapter show how to write more condensed makefiles.)

If you had the above **makefile** and **a.c**, **b.c**, and **c.c** in your working directory and you typed

```
$ make
```

the following would be generated, assuming the **.c** files were error-free

```
cc -c a.c
cc -c b.c
cc -c c.c
cc a.o b.o c.o
a.out
```

and were followed by the result of executing **a.out**.

make's first step is to find its final target. The final target is **a.out**, since **a.out** is the first file mentioned in the leftmost column of the makefile. The construction **a.out: a.o b.o c.o** indicates that before the commands listed under **a.out** can be executed, those under **a.o** , **b.o**, and **c.o** must be executed, if **a.o**, **b.o**, and **c.o** are not up to date. **make** searches the working directory for when the object files were last processed. The files do not even exist, so **make** proceeds to where those files are listed as targets. It sees that each of the **.o** files is dependent on its repective **.c** file. To produce the **.o** files from the **.c** files, **make** must compile the **.c** files with the **-c** option. **make** finds the **.c** files in the working directory, and compiles them. Note that the commands to compile them are listed in the makefile, and are also the first three command lines printed on the screen after you type **make:**

```
cc -c a.c
cc -c b.c
cc -c c.c
```

Having produced the **.o** files, **make** returns to the line

```
a.out: a.o b.o c.o
```

and executes the two commands that are associated with the **a.out** target: **cc a.o b.o c.o**, and **a.out.**

If you type **make** again without editing any of the **.c** files, you will get the message

```
'a.out' is up-to-date
```

If you edit only one of the files and then type **make**, only the edited file and any files that are dependent on it will be recompiled. For example, if you edit **a.c** then invoke **make**, the following commands will be executed:

```
cc -c a.c
cc a.o b.o c.o
a.out
```

The remainder of this chapter gives details on the functions and abilities of this powerful utility.

The make Command

The syntax of **make** is

```
$ make [flags] [macro definitions] [targets]
```

Flags are options. The flags permitted on the **make** command line are shown in Table 3-3. Targets is the title of the commands below it, and often is the name of the file which those commands create. If no targets are mentioned on the **make** command line, the first target in the **makefile** is produced. Macro definitions relates to what is inside your makefile. They are described later in this section entitled "Macro Definitions."

When you issue the **make** command, **make** first reads the flags and puts them into an environment variable called **MAKEFLAGS**. Then, if **make** is called without the **-f** flag, it searches for a file called **makefile** or **Makefile**. If the **-f** flag is used, **make** searches for a file with a name that is specified immediately after the **-f** flag.

Next, **make** reads the command line macro definitions, the other environment variables, and then the specified command line targets. Finally, **make** executes the makefile.

The Makefile

The **makefile** can contain target and dependency specifications, comments, macro definitions, suffix rules, a few special statements, and commands that produce targets. These are described below.

Targets and their Production

The syntax of a target specification is

```
target1 [target2 target3...] : [dep1
     dep2...] [; command]
     [<tab> command] [; command ...]
     [<tab> command] [; command ...]
[#comment]
```

where target1 is the title of the commands that follow. It may be either a file which those commands produce or a symbolic name which is associated with those commands. More than one target may be specified. dep1 dep2 ... indicates dependency files, which

are usually listed as targets elsewhere in a makefile. Up to date versions of the target (dep1) to the right of the colon must be produced before the target on the left of the colon.) In the first example, **a.out**, **a.o**, **b.o**, and **c.o** are targets. The latter three are also dependency files.

Table 3-3. Flags Used with the make Command

Flag	Meaning
-i	Ignores error codes.
-k	Terminates work on the current file and its dependents, if an error status is returned, but continues work on other targets in the makefile.
-s	Does not print commands before executing.
-r	Does not use built-in rules.
-n	Lists, but does not execute commands.
-t file	Touches the target file. The target's date is changed but its contents are not. This option is used only when the file has not been changed since it was last compiled or a neutral change was made (such as adding a comment).
-q	Stands for "question." It checks whether the file is up-to-date, and returns a zero status if yes and nonzero otherwise.
-p	Prints a complete set of macro definitions and target dependencies.
-b	Ensures compatibility with makefiles designed for older versions of the **make** utility.
-d	Invokes the debug mode and, prints very detailed information on files. This option is recommended as a last resort; the **-n** option usually serves best to help you locate makefile bugs.
-f file	Reads file as the description file instead of **Makefile** or **makefile**.

A command may follow dependency information and a semicolon, another command and a semicolon, or a RETURN and a tab. Several commands may be specified under one target. Comments are preceded by a **#** and ended by a RETURN. Whenever anything but a tab occurs in the first eight columns of a line, assumes that the group of commands for the last target is complete. Do not place comments between a target specification and the commands that make the target, or within a group of commands.

Unless an alternative is specified on the command line, only the first target listed in the makefile is produced. For instance, if you alter the previous example so the first two lines read

```
a.o:a.c
     cc -c a.c
```

only **a.o** will be produced; **a.out** will not be produced unless you specify it on the **make** command line

```
$ make a.out
```

If a source code file has an **include** statement, the object code version is dependent on the included file and this dependency should be stated. The source code file is not dependent on the included file.

The **make** command executes each command line in the makefile with a separate invocation of the shell. Hence, if you use the **cd** command in one target command line, the next target command line will still be executed from the original directory.

If a file must be created but there are no directions specifying how to create it, **make** uses the commands associated either with **.DEFAULT** or with suffixes that are dependent on **.SUFFIXES.** The use of these keywords is explained below. If you do not use **.DEFAULT** or **.SUFFIXES**, **make** prints a message and stops.

Macro Definitions: Macro definitions are a convenient tool for changing the files or commands that are used without locating each occurrence of the files or commands in the makefile. Macros are defined with the following syntax:

MACNAM = [string1 string2...]

where the string to the left of the equal sign is replaced by the string to the right wherever MACNAM is reference in the makefile. If the macro name contains more than one character, the **macro** is referenced in the makefile with the following syntax:

$ (MACNAM)

If the macro name is only one character long, the parentheses may be ommitted. For instance, if the macro name is simply **M**, the macro can be referenced with **$ M**. Here is an example illustrating macros:

```
FILES=a.c b.c c.c
O=a.o b.o c.o
a.out:$ O
     cc $ O
     a.out
$ O: $ (FILES)
     cc -c $ (FILES)
```

The macro definitions can be changed by either editing the first two lines of the makefile, or listing the new macro definitions on the make command line. If the command line definition contains more than one string, it is enclosed with quotes. For example:

```
make FILES = "func1.c func2.c" O="func1.o func2.o"
```

will execute the makefile with the macros redefined accordingly.

One disadvantage of using macros is that **make** does not distinguish between the files defined in the macro. If any file in a given macro needs to be recompiled, **make** recompiles all the files in that macro. The extra time spent in these compilations may or may not be compensated for by the convenience of using macros. It all depends on your individual files and how much system time you can afford.

If the final target of a makefile is a macro that defines two or more files, only the first file will be produced. To avoid this problem, the final target should be a symbolic name with a dependency file that is the macro.

Suffix Rules: In creating files, **make** consults a file, **.SUFFIXES**, which includes certain default suffix transformation rules. When a **.o** file without directions on how to produce the file occurs, **make** consults default rule in **.SUFFIXES** for producing a **.o** file. This rule is: Search the current directory for a file with a base name that is the same. If its suffix is **.c**, execute **cc -c file.c. make** does this for each of the suffixes and compilers listed in Table 3-4.

Table 3-4. Implicit Rules for Producing .o Files

Suffix	Compiled by
.c	C compiler
.s	Assembler
.l	Lex compiler
.y	Yacc-C compiler

You may specify rules that override or add to the default rules. Here is an example that uses **.SUFFIXES** and defines a rule for processing **nroff** source files:

```
OBJS=1.doc 2.doc 3.doc
.SUFFIXES:  .doc .me
.me.doc:
     nroff -me $*.me > $@
document: $(OBJS)
     cat $(OBJS) > document
```

The **.SUFFIXES:** line warns that all files with suffixes **.me** and **.doc** are to be treated specially. Suffixes must be listed here in the opposite order from which they will be used. For instance, if a **.b** file is to be transformed into a **.g** file, and the **.g** file into a **.w** file, you should list, from last to first:

```
.SUFFIXES: .w .g .b
```

The **.me.doc** line indicates that files with the suffix **.me** are to be transformed to files with the suffix **.doc**, according to a rule that will be given on the next line(s) of the makefile. Order is also important here, and is the opposite of the order required by **.SUFFIXES:** The file with the first suffix listed is transformed into a file with the second suffix. The transformation rule in the above example is: **nroff** any file with a suffix **.me**, and place the output into another file that has the same root or base name but ends in **.doc**. The special symbols **$*** and **$@** are explained in the next section.

Internal Macros: The **make** tool supplies several internal macros for use with suffix rules and dependency declarations. These macros are described in Table 3-5.

Table 3-5. Internal Macros Used with the **make** Tool

Symbol	Meaning	Used with: Files (file1: file2)	Used with: Suffixes (.c.o:)	Example
$*	The filename of the current dependent without the suffix.		x	**$*.c= file.c**
$@	The full name of the current target.	x	x	**file1 and file.o**
$<	The full name of the file on which the current target depends.		x	**file.c**
$?	All out-of-date files on which the current target depends.	x		out-of-date **file2**
$%	An archive library file of the form lib(file.o).	x		$@($%)= lib(file.o)

This example illustrates the use of .SUFFIXES and internal macros.

```
.SUFFIXES: .o .c
.c.o:
     cc -c -o $@ $*.c
ex: a.o b.o c.o
     cc -o ex a.o b.o c.o
     ex
```

Two additional internal macros are useful when working with more than one directory: D and F. They are used in conjunction with the macros mentioned above and a macro that you declare in your makefile. This macro contains a directory and filename, in the conventional syntax directory/filename. An example of these macros is

```
DIR_MODS= d1/targ1  d2/targ2
all: $(DIR_MODS)
     @ echo 'everything's up to date'
$(DIR_MODS):
     cd $(@D); $(MAKE) $(@F)
```

include: Makefiles may contain **include** statements. The syntax is

include<space or tab>string

where **string** is the name of the file to be included. Up to 16 nested **include**'s are permitted. That is, an **include** may have a file that includes another file and so on up to 16 times.

Maintaining Archive Libraries: **make** has a special command that helps to maintain archive libraries. The syntax for maintaining a library named **lib** is

```
lib(fname.o) lib(fname2.o)
@ echo lib updated.
```

This command is included in the makefile. The syntax **lib(file1.o file2.o)** is not legal.

Recursive makes: The **make** command may call itself. To invoke **make** from within a makefile, you use the internally defined macro **$(MAKE)**. For example:

```
xx:
     $(MAKE) -f newmakefile [macro defs] [targets]
```

Part of **make**'s internal knowledge is that the macro **$(MAKE)** means **make**. **$(MAKE)** is the only command executed when the **-n** option is in effect. Using the **-n** option on the **make** command line instructs **make** to list the commands it would execute but not to execute them. The **-n** option is exported to the subsequent **make**, so its commands are also listed without being executed. This option allows you to see the steps that would be taken by the **make** call without waiting for the system to perform those steps. You can use **$(MAKE)** as many times as you like during a program, executing an unlimited number of makefiles.

Compatibility with Old Makefiles: This document describes the augmented version of **make**. To ensure compatibility with old makefiles, use the **-b** option on the command line.

Special **make** Statements

Several special statements can be used in a makefile. Some of the results of these statements can also be invoked on the **make** command line.

.SILENT: Each command in the makefile is printed as it is executed. Three ways are available to silence the printing of commands:

- Globally, by inserting **.SILENT** into your **makefile.**
- Individually, by inserting a **@** between the tab and the command in each command line that should not be printed.
- Globally, by using the **-s** option on the **make** command line.

.IGNORE: Normally, **make** terminates if any command returns a nonzero (error) status. You can prevent this termination three ways:

- Individually, by placing a hyphen after the tab and before the command line words in each line with an error status that is to be ignored.
- Globally, by inserting **.IGNORE** anywhere in the makefile.
- Globally, by using the **-i** option on the **make** command line.

An alternative to the **-i** option is **-k.** This option terminates work on the current line and its dependents in case of error status, but continues work on the nondependent lines in the makefile.

.PRECIOUS: Pressing the **Cancel** key during **make**'s execution stops all work. Any file being processed is removed unless it has been declared a dependent of **.PRECIOUS** in the makefile. Files on which **.PRECIOUS** is dependent are not removed.

.DEFAULT: When a file depends on a nonexistent file that **make** does not know how to create, **make** looks for a line labeled **.DEFAULT** and executes the commands listed there. For example:

```
.DEFAULT
        cc -o georges 1.c
niceone: georges
        georges | cat
```

In this example, since **make** cannot find a rule to make **georges**, it follows the rule under **.DEFAULT.**

4 Examining Source and Object Code

During a programming task, you may want to know the size and names of your object files and the location of functions in your source files. The following FOR:PRO tools allow you to examine your code for this information.

size: DETERMINING THE SIZE OF AN OBJECT FILE

The **size** tool displays the size of an object file in bytes. It prints the decimal number of bytes required by text, data, and bss portions. It also prints the sum in hexadecimal and decimal of each object file argument.

The **size** utility uses the name of the object file that you are measuring. If you do not specify a file, **a.out** is used.

The syntax for **size** is

```
$ size  filename
```

For example, to see the size of a program named **test.o**, you enter the following:

```
$ size test.o
```

The result is

```
text    data    bss    dec    hex
60      16      0      76     4c
```

name (nm): EXAMINING THE SYMBOL TABLE NAMES OF OBJECT FILES

The **nm** command tool prints the name list (symbol table) of each object file in the argument list. If an argument is an archive, a listing for each object module in the archive is produced. If no file is given, the symbols in **a.out** are listed.

In the name list produced by **nm**, each symbol name is preceded by its value (blanks if undefined) and one of the following letters:

U	Undefined	B	Bss segment symbol
A	Absolute	C	Common symbol
T	Text segment symbol	F	File name
D	Data segment symbol	-	For fdb symbol table entries

For local symbols (nonexternal), the type letter is in lowercase. Output is normally sorted alphabetically.

The syntax for **name** is

```
$ nm [-option] filename
```

In this command, the options control the type of listing that is produced. The options are shown in Table 4-1.

Table 4-1. Options for Listing Names of Files

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

Sample Program for Use with the name Tool

The following C program illustrates the effect of **name**.

```
/*Find the position of the first occurrence of the
* character z in a string and return its position in
* the string.
* Return 0 if the character z is not in the string.
*/
```

```
#define NONE  0
char search_chr = 'z';
main()
{
     short index;

     index = position("zorro",search_chr);
     printf("the position of z in this string = %d\n",index);
}

int position(string,c)
register char *stgring,c;
{
int i = 1;
     do {
          if (*string == c)
               return(i);

          i++;
     }while (*string++);
     return(NONE);
}
```

Compile this program using the **-c** option to produce an object file **pos.o**:

```
$ cc -c pos.c
```

Typing **nm pos.o** displays the following name list on your screen:

```
         U _csavl
         U _regsav
00000000         T main
00000060         T position
         U printf
000000bc         D search_chr
```

ctags: DETERMINING THE LOCATIONS OF FUNCTIONS IN SOURCE FILES

This command creates a **tags** file for cross reference purposes. This **tags** file gives the locations (line numbers) of specified functions in C, Pascal, and FORTRAN source programs. Each line of the **tags** file contains three fields:

- Function name
- File in which the function is defined (line number)
- Scanning pattern used to find the function definition

Files ending in **.c** or **.h** are assumed to be C source files and are searched for C routines and macro definitions. Others are first examined to see if they contain any Pascal or FORTRAN routine definitions. If not, they are processed again looking for C definitions.

In C programs, a special main tag can be used to make **ctags** practical in directories with more than one program. The main tag is created by adding an **M** to the beginning of the filename and removing the **.c** from the end. The leading pathname components are also removed.

The syntax for **ctags** is

```
$ ctags [-options] filename
```

The ctags command is used with the options shown in Table 4-2.

Table 4-2. Options for Use with the ctags Command

Options	Descriptions
-u	This update option replaces the existing **tags** file with a new **tags** file.
-a	Adds information to the end of a **tags** file.
-x	Displays the **tags** file with line numbers.
-w	Suppresses warnings.

Note that only the **-x** option displays the **tags** file on the screen. To view a **tags** file created by the other options, you must use either the **more** command or the **cat** command.

Sample Program for Use with the ctags Tool

The following program illustrates the effect of **ctags**:

```
main()
{
short int x;

     inc(x);
     sqr(x);
     dec(x);
     neg(x);
}
```

```
inc(num)
register short num;
{
     num++;
}

sqr(num)
register short num;
{
     num *= num ;
}

dec(num)
register short num;
{
     num--;
}

neg(num)
register short num;
{
     num = -num;
}
```

If you type **ctags -x filename**, the output will be

```
dec      23 ctags.c    dec(num)
inc      11 ctags.c    inc(num)
main      1 ctags.c    main()
neg      29 ctags.c    neg(num)
sqr      17 ctags.c    squ(num)
```

5 DEBUGGING PROGRAMS

The Language Development Tools provide a Fortune Systems symbolic debugger (**fdb**). This is a high level debugging tool with which to debug programs in terms of their source-level names and constructs. Since **fdb** is language independent, it can serve as a debugger for the three high level system languages supported on the Fortune System--C, FORTRAN, and Pascal.

PREPARING A PROGRAM FOR DEBUGGING

The **fdb** debugger is used to debug programs after they have been compiled with the **-g** debug option, and loaded. To prepare the executable programs to work with the debugger, you must do the following:

- Compile source programs with the **-g** option
- Run the loader (if needed) with the **-l** option
- Run **fdb**

RUNNING fdb

The format of the **fdb** command is

```
$ fdb [objectfile]  [directory]
```

In this command, **objectfile** is an executable program file that has been compiled with the **-g** (debug) option. The default for **objectfile** is **a.out**. The **directory** is the directory where the source files exist. The default for **directory** is the working directory. Note that **fdb** uses * as a prompt character to indicate that it is ready to accept a command. You can change the debugger prompt with the **set** command.

The following example illustrates the commands to compile a C program and a Pascal program, to link the programs, and to call the debugger.

$ cc -c -g test.c
Compiles the program **test.c** and leaves output in **test.o**.

$ pc -c -g sample.p
Compiles the program **sample.p** and leaves output in **sample.o.**

$ cc -o sample.obj -g test.o sample.o
Links the two object programs.

$ fdb sample.obj
Calls the debugger.

The computer responds:

```
Fortune Symbolic Debugger
*     (Indicates fdb is waiting for your command.)
```

An extended example of a debugging session is provided in the individual language guides for C, FORTRAN, and Pascal. This chapter provides a more complete description of the fdb commands.

SPECIAL fdb RULES

Note that you must observe a number of rules when using **fdb.** These rules are explained under the next few headings.

Uppercase and Lowercase Rule

Uppercase and lowercase letters are generally treated the same. Combinations of uppercase and lowercase letters are allowed. This rule only applies to the **fdb** keywords.

For example, the following commands mean the same thing:

equate
EQUATE
EquAte

Variable and procedure names, however, may have distinct uppercase and lowercase letters that must be respected.

Abbreviation of Commands

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 **equate**

Some commands can even be abbreviated to one character (see Table 5-1). However, if a command is not given in one or three characters, the whole command string must be spelled out. For example, for the **equate** command, **eq**, **equal**, and **equat** are illegal, while **e** and **equ** are legal.

Use of Leading Blanks

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

```
EQUATE
eQU
Equate
```

Multiple Commands per Line

Multiple commands per line are allowed if they are separated by a semicolon. Since the commands in multiple-command line entries are interpreted serially, the first command is performed regardless of the error condition in 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, a **comment** command, or a **break-do** command. For example, each of the following lines is a single command:

```
find  "a=0;  b=0;  c=0;"
equate a  ",a/wx; ,b/c; , c;  break when count=100;"
comment  x:=3; was for Pascal assignment.
break  3 do display a;  display b;  sho file
```

Source-File Dependency

The **fdb** debugger generates a warning if the source file for the debugged object file does not exist in the directory from which the object file was compiled. It is also possible that the source file and the object file to be debugged are not the same version. The fdb debugger warns you if the source file is newer than the object file.

Null Procedure Name

When the procedure name isn't given for a C program, **fdb** assumes the main procedure name since the main entry point is defined by the main () procedure in the C language. The main procedure convention isn't required in FORTRAN and Pascal. Therefore, to use a consistent main procedure reference, **fdb** treats a null procedure name as the main program name. For example:

break :3 Sets the breakpoint at line 3 of the main program.

display :a Displays the variable a in the main program.

Use of Quotes

A quote in a string is represented by two quotes. The quote **abc"d** is represented by **abc""d**. The string **""** is represented by **""""**, but **"""** is an illegal string.

Use of the Backslash

A backslash \ is used to indicate that a special character follows. Therefore, \\ means single \. Use a backslash whenever non-alphanumeric characters are used. (This rule does not apply to the **alias** replacement string.) In general, a backslash before a special character suppresses the special interpretation of that character.

For example, if \ precedes **%**, the **equ** expansion is suppressed:

```
equ a "XYZ"
find "%A"      Searches for YZ

find "\%A"     Searches for %A
```

Use of the Carriage Return

When using the debugger, pressing the **Return** key immediately after the prompt is interpreted as re-entry of the previous command. If the previous command was a multiple command line, the carriage return executes the last command on the line.

USING fdb COMMANDS

The debugger uses three types of commands: (1) those that allow you to set up the debugging environment or to display information; (2) those used for source file examination; and (3) those that control execution. Table 5-1 summarizes these commands.

Table 5-1. The Debugger Commands

Command	Purpose
alias(a)	Defines a debugger command
break(b)	Sets a breakpoint
comment	Prints a comment line
delete(d)	Removes a breakpoint
display(,)	Displays the value of a variable
dump	Displays the contents of memory
equate(e)	Equates a character to a data string
file	Redefines source, input and output files
find(f)	Searches for a specified string
fun	Defines the function keys
go(g)	Starts or resumes execution of the program
help(h)	Lists every **fdb** command
let(l)	Assigns a value to a variable
print(p)	Prints a number of lines
quit(q)	Exits the debugger
restart	Restarts the program
set	Sets up the debug options
shell escape(!)	Executes shell commands
show(s)	Shows information about session
trace(t)	Traces the execution
variable address(&)	Displays a variable address
walk(w)	Single-steps through code

The syntax for each of these commands is described in the following pages, together with examples. In these descriptions, all punctuation (except square brackets), such as commas, parentheses, semicolons, hyphens, or equal signs, must be included as shown.

alias: Defining Debugger Commands

The **alias** command allows a user to define debugger commands or cancel pre-set **alias** commands. You can also use **alias** to rename existing **fdb** commands or to combine two or more commands into one for convenience.

The syntax for **alias** is

alias definition [replace string]

To see the current **alias** definitions, type

show alias

You must cancel one **alias** definition before re-defining another. An **alias** definition can be canceled by typing the empty replacement string as follows:

alias definition

If multiple commands are used in the **definition**, they must be enclosed in quotes. Note that the case rule does not apply to the definition string.

The following examples redefine the commands **single** and **step**:

alias single walk	Defines single as walk
alias step "walk; display a"	Defines step as walk and then displays a line
alias single	Cancels the alias for single

break: Setting a Breakpoint

This command sets a breakpoint at the indicated line number in the source program. The program is stopped before the indicated line is executed. If this line is not an executable statement (such as a blank, a comment line, or a declaration), the breakpoint is set to the first executable line that follows. The syntax for **break** is

break[procedurename:][line number] [do(fdb command)]

The module name and/or line number may be omitted in the **break** command. In this case, the defaults are taken from the current procedure name and current line number. If the breakpoint is to be set in a procedure other than the currently active procedure, a line number must be specified. If the line number is smaller than the procedure starting line number, then the breakpoint is set at the procedure starting line number.

If the **break** command is specified with a **do** phrase, **fdb** will execute the command(s) when the breakpoint is reached. Otherwise, control is transferred to the user. For example:

break	Sets a breakpoint at current line in the current procedure.
bre sub1: 1	Sets a breakpoint at the first line in procedure SUB1.
break 10 DO ,a; ,b	Sets a breakpoint at line 10 and prints the values of variables a and b whenever the program stops.

comment: Printing a Comment Line

This command causes **fdb** to print a comment line exactly as entered. The syntax is

comment statement

The comment command is used to document the debugging session, especially in cases where the **fdb** output is saved in an external file.

The following example is used to test for error conditions.

com;; equ a;

It is one command even though three semicolons appear.

delete: Removing Breakpoints

The **delete** command is used to remove breakpoints. Its syntax is

delete [procname:] [line number]

The **del all** command deletes all the breakpoints set so far. The **del** command with no parameters deletes breakpoints interactively. That is, each breakpoint location is printed, and the user can decide if the breakpoint should stay. The user responds **d del, y, yes,** or **ok** to delete the breakpoint. All other responses are interpreted as **no** responses. For example:

delete main:4	Deletes the breakpoint on line 4 of procedure main.
delete all	Deletes all the breakpoints.

display: Displaying the Value of a Variable[]

This command is used to display the value of a variable at the point of program suspension. Its syntax is

display [proc name:] variable [format]

The values are displayed in a format determined by the user. If the format specification is omitted, variables are formatted according to their data type (as declared in the program).

The comma is used as an abbreviation for **display**. Two commas **,,** display the most recently displayed variables. The contents of pointer variables can be displayed by **->** in C and by **^** in Pascal.

Registers can be displayed by prefacing the register name with a **<** sign. For example, the contents of register **d0** in hexadecimal can be displayed by typing display or **g <d0/x**.

The contents of an absolute address can also be displayed. For example, typing **display** or **g 0x234b/x** yields the contents in hexadecimal of that location.

Array variables can be specified as a range to display more than one element at a time.

Variable Format Specifications: Displaying Variables and their Contents

Format specifications can be used for displaying variable contents using this syntax:

/lm (l=length and m=format)

The format specifiers for length are

b	One byte.
h	Two bytes(half word).
l	Four bytes(long word).
number	String length for formats **s** and **a**.

The format specifiers for format are

c	Character.
d	Decimal.
f	Floating point.
g	Floating point.

o Octal.

s Assume variable is a string pointer, and print characters until a null is reached.

u Decimal.

x Hexadecimal.

The length specifiers are only effective with formats **d**, **u**, **o** and **x**. If one of these formats is specified and the length specifier is omitted, the length defaults to four bytes.

In the following example, the types and contents of variables i, p, a, and j are defined as:

Variable Type	Name	Contents
char	i	'x'
char	*p	"abcxy"
char	a[3]	"ABC"
int	j	0x12345678

The following commands illustrate several ways to examine the variables listed above:

Command	Displays
display i	x
display i/x	0x78000000
display p	abcxy
display p->/c	a in C (should be **p^/c** in Pascal)
display p/3s	abc
display p/s	abcxy
display a	ABC
display a/2s	AB
display j	305419896
display j/x	0x12345678
display j/b	18

dump: Displaying the Contents of Memory

This command displays the contents of memory in character format(**c**) or in hexadecimal format(**x**). (The default is hexadecimal.) The syntax for **dump** is

dump [option] address

The output format for each line is as follows:

- Space designation: I for instruction space or 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, two lines are displayed.

For example:

dump 0x100	Dumps between 0x100 and 0x10f.
dump 3	Dumps between 0x0 and 0xf.
dump 0x100 / 0x200	Dumps between 0x100 and 0x200.
dump next	Dumps next 16 bytes.
dump &a	Dumps the memory around the address of variable a.

equate: Using a Character for a Data String

The **equate** command equates a character to a data string. The syntax is

equate definition [replacement string]

To expand the meaning of an equated character, type the escape character **%** before the equated character and then type the addition to the data string. The equated character will be expanded in line prior to the execution of the command.

The **equate** command can also combine multiple commands into one command or into alias commands.

You can cancel an **equate** command by equating the previously defined character to a null (empty) string. The **fdb** debugger detects and reports recursive **equate** definitions.

For example, to equate a character to a long variable name use the following:

equ a "employee"	Defines "a" as equated to "employee".
display %a	Displays the contents of variable employee.
display %a.name	Displays the contents of variable employee.name.
equ a	Cancels the equate definition.

file: Re-defining Files

This command re-defines the source file or re-directs input and output from the standard devices. The syntax of **file** is

file [<>] [filename]

Files for the program being debugged can be redirected by using runtime arguments as described in the section on the **restart** command.

If a filename does not include a dot, it is considered a function name, and the file with that function becomes the source file. The symbols < and > are used with the **file** command to redirect fdb's input and output. A filename immediately following < will cause input to be read from that file. A filename with a following > will cause the output to be redirected into that file. When you type **file**, followed by < or > and a space, **fdb** will direct the debugger input or output to the standard devices. Use >> to append to the end of an existing file. It is important to have a file command to redirect input at the end of any file that you use as a source of input to fdb.

For example:

file <profile	Executes fdb commands in profile.
file /usr/source/test.c	Makes source file /usr/source/test.c.
file > y/trace	Saves debug output in parent's directory.
file >	Redirects debug output to the terminal.

find: Searching for Strings

The **find** command is used to search the current source file for a specified string. After searching, it then prints the source lines that contain the specified string. The syntax is

find [string] [line count]

The search can be confined to a specific section of the program by specifying the beginning line number or ending line number immediately following the strings. The debugger can also be instructed to find multiple occurrences of the specified string by typing ! followed by a decimal value for the number of occurrences needed.

To search backward, use the command:

find .! -1

The **-1** means "search backwards." If the string is not specified, the last search string is used.

The default value for the line number is the current line. The current line number is automatically updated each time a line is found with the specified string. The line count defines the maximum number of lines to be printed.

Examples of the use of **find** are as follows:

find "procedure"	Searches for "procedure" and prints the first line that contains the string from the current line.
find "if" 3	Finds the first "if" beginning at line 3.
find "count" 2 ! 10	Finds 10 ocurrences of "count" starting at line 2.
find "xyz" 10 / 100	Finds "xyz" string starting at line 10 and searching through line 100.

fun: Defining the Function Keys for Use with fdb

This command allows the user to define the 26 function keys with useful **fdb** commands. All the function keys, F1 through F16 on the top of the keyboard, the Help and Execute keys on the left and right sides of the keyboard, the four grey cursor keys on the right side of the keyboard, and the four keys above the cursor keys (Insert, Delete, Prev Scrn, Next Scrn), can be programmed (or "aliased") by the user. For more information on programming the function keys, see Table 5-2, and the sections "alias, "Set" and "show" in this chapter. The proper syntax is

fun key description

Since the first letter of the function key name can be defined as uppercase, you can define 52 function keys. If multiple commands are defined by a function key, the function description must be enclosed in quotes.

Table 5-2 shows the function keys that are pre-defined by **fdb**. You may re-define these function keys. The command **show function** displays all the current function key definitions.

Table 5-2. Function Keys Pre-defined by fdb

Key	Definition	Description
help	Same as **Help** command	See **Help**
fun16	**Next**	Can be used with **print** or **dump**
execute	**Walk**	May go to subroutine
EXECUTE	**Walk in**	Allows single step with same procedure
prev scrn	**Print . ! -20**	Prints previous 20 lines
next scrn	**Print .-1 ! 20**	Prints next 20 lines
up arrow	**Print . -1**	Prints previous line
down arrow	**Print next**	Prints next line

Two examples for using this command are

fun 1 ,employee.name	Same as FUN fun1 ,employee.name.
fun Fun1 "walk; ,,"	Shifted **Function1** key is defined as a single step and displays the contents of the last variable referenced.

go: Resuming Program Execution

The **go** command is used to continue program execution or to continue with the signal that caused the program to stop. The command causes the program to either start or resume execution. The syntax for **go** is

go [sig]

If the **sig** option is used, the program continues with the signal that caused it to stop. This option can be used to debug a user signal handler.

The program continues to execute until a breakpoint, a program error, a user interrupt, or a normal program exit occurs.

For example:

go sig	Continues with the signal that caused the suspension of program execution.
go	Continues from the current line.

help: Viewing the Debugger Commands

This command lists every **fdb** command with a short description of the command. The syntax is simply

help

You may also use the **Help** key or a single "?".

let: Assigning a Value to a Variable

This command is used to assign a value to a variable. The syntax is

let assignment statement

For C and FORTRAN, the assignment operator is =; for Pascal the assignment operator is :=. For example:

let a = 'c'	Assigns character c to variable a.
let b = '\007'	Assigns beep character to variable b.
let c = 1.23e3	Assigns real value of 1230.0 to variable c.

print: Printing Program Lines

This command prints a specified number of lines from a given starting line in the source code. The syntax is

print [current line] [number of lines]

The default value for the starting line is the current line. The current print line is changed, but the current execution (or walk) line is not affected. For example:

prints	Prints the current line.
print .-10 / 11	Prints 10 lines before the current line through line 11.
print . ! 6	Prints 6 lines from the current line.

quit: Exiting the Debugger

The **quit** command causes an exit from the **fdb** debugger. The syntax is simply

quit

You are returned to the shell, and the debugging session ends.

restart: Restarting the Program with Runtime Parameters

You can use this command to restart the program being debugged with runtime parameters. The syntax is

restart [options] [(parameters)]

Options and parameters for the program being debugged can be set up. This is the difference between the **restart** and **go** commands. The standard input/output device for the program being debugged can be redirected by using > or < command line arguments just like if you were in the shell.

For example, to debug an object file called **f.o**, with option **-o** and parameter **input1**, use the following commands:

```
$ fdb f.o Debugs the object file f.o
```

The computer responds:

```
Fortune Symbolic Debugger
*
```

Then you type:

restart -o input1 Invokes option **-o** and parameter **input1** for the object file **f.o** when typed after the above command.

set: Setting Debug Options

This command is used to set the debug options. The syntax is

set option [=] definition

The four debug options are as follows:

prompt Changes the **fdb** default prompt. The default is *****, but any string can be used.

case Makes **fdb** differentiate uppercase and lowercase letters for variable and procedure names. The default is upper/lower (**UPLOW**). The other options are **upper** and **lower.**

language Changes source language convention. The default is the actual source language used (C, FORTRAN, or Pascal).

mode Suppresses **fdb** error messages. In case of user error, the beep will sound, and mode will be set to **terse.** The default is **verbose.**

Examples of the **set** command are:

set prompt = "+"	Sets debugger prompt to +.
set prompt="Fortune fdb%"	Sets prompt to Fortune fdb%.
set case upper	Converts variable and procedure names to uppercase.
set language C	Indicates user should use C language convention.

shell escape (!): Executing Shell Commands while Debugging

This command allows the user to execute shell commands in the middle of a debugging session. The syntax is

! command

Multiple shell commands on a single line are not permitted in **fdb**.

Examples of the use of the shell escape include:

!date	Prints date and time.
!date; !who	Executes multiple shell commands who from fdb.
!date; who	Is illegal since multiple shell commands from the command line are not allowed.

show: Displaying Information About the Debugging Session

This command displays information about the current debugger session at you terminal. The syntax is

show command

The information that can be displayed is shown below. Note that all of these commands can be abbreviated to three characters.

Command	Displays
alias	Alias definitions.
breakpoint	Breakpoints that are currently set.
case	Uppercase or lowercase, whichever is in use.
command	The last command as seen by fdb (expanded in case of alias).
equ	A list of all equate symbols and their definitions.
file	Input/output/source files.
fun	Function key definitions.
language	Source language specification.
mode	**terse** or **verbose**, whichever is in use.

procedure A procedure stack (the procedure names called to reach the current stop point).

window A few lines around the current line--default is five lines on each side of the current line.

The following are examples of the **show** command:

show procedure Lists procedure names in frame stack.

show breakpoint Shows all the breakpoints defined.

show equate Shows all the equate definitions.

show window 4 Prints four lines above and below the current line.

trace: Tracing the Execution of a Program

This command traces the execution of the source program. There are three types of **trace** commands: trace execution, trace procedure, and trace variable var.

Trace execution is used to display the code-segment labels (code statement line numbers) encountered during program execution. The source lines will also be printed.

Trace procedure prints source lines whenever a new function (procedure or subroutine) is entered. The output format is

****AT line # called: func name (argument name= argument value) [FILE= source file name, LINE= line # in the file, NEST= nest level]

Trace variable var prints source lines whenever the specified variable changes its value. The variable must be defined when it is used. For example, an automatic variable cannot be traced outside the procedure in which it is defined.

&: Displaying the Address of a Variable

This command is used to display the address of a variable. The syntax is

& variable name

The address is always displayed in hexadecimal format. For example:

&a Displays address of variable a.

&b[3] Displays address of the 4th element of array b.

&b(3) Displays address of the 3rd element of array b. This format applies to FORTRAN. For Pascal and C use [] as shown above.

walk: Single-Stepping through a Section of Code

The **walk** command is used for single-stepping through a section of code. The syntax is

walk [| in | out]

The number of statements to single-step can be specified by a parameter. The user can **walk** within a current module (**walk in**) or into the called procedure (**walk out**). The default is **out**.

The following are three sample lines of source code that can be used to illustrate the **walk** command:

```
line 10  count = 10;
line 11  getvalue();
line 12  printf("result=%d0, count);
```

For a user who "walks" in the source code at line 10, **walk**, **walk in** and **walk out** are equivalent. The variable count is set to 10, and execution is stopped at line 11.

At line 11, **walk in** will execute the **getvalue** procedure and stop at line 12; **walk out** will stop at the first line in the **getvalue** procedure.

At line 12, **walk out** has no meaning in the nonsystems programming environment. The **fdb** debugger will not single-step the **printf** routine. The commands **walk in** and **walk out** are thus equivalent.

SPECIAL CHARACTERS

The following characters have special meanings in fdb:

Symbol	Meaning
,	A substitute for the **fdb** command **display** or the most recently referenced variable name.

&	The address of a variable.
!	The shell escape or the counter in the **print** and **find** commands.
?	The **help** command.
.	The current line (in **print, find, ...**).
:	The procedure name identifier (or character that terminates procedure names).
%	The **Equate** key.
\	The **escape** character.
;	New line end of single command.
/	A symbol for giving a line range (for **print** or **find**), or marking the start of a format specification.
<	The register display indicator.

DEBUGGER MESSAGES

Each **fdb** message consists of three parts: type, code, and reason for message. These appear in the following format.

The type refers to five different categories of messages. These are:

INFORM:	Nothing wrong; just for information
WARNING:	Not critical but not desirable
ERROR:	User error; commands are ignored
SYSTEM:	Reached system limit or **fdb** has a bug
PROCESS:	Fatal error, so there is no reason to continue the **fdb** session

The code refers to the internal sequence number for documenting errors. The @ symbol in the following messages is replaced by a specific string. The messages are as follows:

Code	Description
0	Starts fdb
1	Object file @ does not exist in directory
2	Object file @ is newer than core
3	No core image exists
4	No string table exists for object file @
5	No room for @ bytes of string table
6	I/O error in reading string table from object file @
7	No room for @ bytes of symbol table
8	I/O error in reading symbol table from object file @
9	String index in symbol table messed up
10	Source file @ does not exist in directory
11	Source file @ newer than object file @
12	Obj file @ not compiled with -g option
13	Interrupt requested, (message 0 follows)
14	Illegal instruction, (message 0 follows)
15	Illegal command @
16	Yow! Memory ran out in allocating for @
17	Core dumped
18	This module not loaded with -g option
19	Program ready to execute
20	(Reserved for future use)
21	Core access error at @ in @ space
22	(Reserved for future use)
23	Cannot file line number from object file
24	Input device directed to @
25	Output device directed to @
26	Multiple input device definition
27	Multiple output device definition
28	Input @ cannot be opened
29	Output @ cannot be opened
30	No breakpoint currently set
31	No breakpoint to delete
32	Cannot set breakpoint there with errno= @
33	Breakpoint already set there
34	Address not found for proc: @ at line no= @
35	Execution suspended due to breakpoint set at @
36	Execution suspended to perform breakpoint command at @
37	Digit expected at @
38	Too long strings used
39	Source language is @
40	Bad data class @
41	Unrecognized source language spec @
42	Recursive alias for equ definition
43	Equ definition character @ must be alpha
44	No delimiter after equ definition character @
45	Equ char for @ must be defined before cancel
46	Equ definition for @ already exists

```
47      Currently no equ defined
48      Equ expansion causes longer than 1024 characters
49      Equ char for @ must be defined before used for expansion @
50      String expected but non-quote encountered @
51      Unmatched quote for @
52      Needs more parameter after @
53      No parameter needed on/after @
54      Syntax error at @
55      No old command to repeat
56      Must precede format specification at @
57      No matched string for (@) from line number @
58      Error in string match
59      (Reserved for future use)
60      Module (procedure) too long @
61      Procedure @ is not active
62      Procedure name @ does not end with a colon
63      Only one level proc name allowed
64      Illegal variable name detected at @
65      Unmatched bracket for array variable @
66      Expected null variable name
67      No variable name found in display command
68      Unrecognized register variable @
69      Procedure call requires ( and )
70      Unrecognized argument at @
71      Unrecognized relational operator @
72      Procedure @ not found in symbol table
73      Unrecognized character in proc argument @
74      More arguments than fdb can handle - max is 16
75      Unknown variable @ - address not found
76      Var @ not found in procedure:
77      Illegal variable descriptor @ used
78      Multiple source file definition
79      Address of register var cannot be displayed
80      Null filename specified at @
81      File @ opened to append
82      File @ opened for output
83      No source file read in memory
84      Line number @ is too big when last line is
85      Illegal line number @
86      Error in line range specification
87      Illegal integer specification at @
88      Argument length @ is too long
89      Bad magic number @
90      Integer too big in converting to string
91      Stack messed up
92      Unrecognized signal @
93      Normal return from proc call
94      Bad file linked list
95      No process found to continue
96      Bad subprocess command @
97      Subprocess cannot be created (fork)
98      Error in ptrace wait
99      Process terminated
```

```
100     @
101     This feature is not supported by current version of fdb
102     Register @ is not valid
104     _dsubc variable not found
105     Main entry program not found
110     Command buffer is empty!  Please try again
111     Symbol table mangled for @
112     Bad linked list for @
113     Bad common block
114     Lost common block
115     Illegal function key of @
116     Currently no function key defined
119     No entry for @ in symbol table
120     Command length or alias string too long - max is 12
121     Alias for @ must be defined before use
122     Alias for @ already defined
123     Currently no alias defined
127     Hex conversion
132     Illegal assignment command at @
133     Incompatible  data type
134     Illegal character data type
```

PLEASE GIVE US YOUR RESPONSE TO THIS MANUAL

You can help us provide manuals that suit your needs by filling out and returning this form. When a new edition of this manual is prepared, we will try to use your suggestions.

Write the name of the manual you are commenting about here _____

_____

1. Does this manual give you the information you need? Yes No
   Is any information missing?

2. Is this manual accurate? Yes No
   Please list the inaccurate information.

3. Is the manual written clearly? Yes No
   What areas are unclear?

4. What other comments about this manual do you have?

5. What do you like about this manual?

On a scale of 1 to 10, how would you rate this manual? Please circle one.
Excellent 10 9 8 7 6 5 4 3 2 1 Poor

Name _____ **Phone number** _____
Company _____
Address _____
City _____ **State** _____ **Zip Code** _____

Fortune Systems Corporation has the right to use or distribute this information as appropriate with no obligation.

1002348-02

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Class Permit No. 29 San Carlos, CA

FORTUNE SYSTEMS CORPORATION

Attn: Publications Department
101 Twin Dolphin Drive
Redwood City, CA 94065

USER RESPONSE CARD

FOR FORTUNE SYSTEMS MANUALS