ULTRIX

digital

Guide to Languages and Programming

Order Number: AA–ML94B–TE

ULTRIX

Guide to Languages and Programming

Order Number: AA-ML94B-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

digital equipment corporation
maynard, massachusetts

Contents

About This Manual

1 Introduction to the Compiler System

2 Storage Mapping (RISC)

3 Storage Mapping (VAX)

4 Language Interfaces (RISC)

5 Language Interfaces (VAX)

6 Improving Program Performance (RISC)

7 Improving Program Performance (VAX)

8 Debugging

11 System Calls and Library Routines with Security Implications

A C Implementation

Examples

Tables

About This Manual

This guide describes the compilers and high-level languages that are part of the ULTRIX compiler system. It also provides an overview of the ULTRIX driver commands and system tools that are provided for this programming environment.

Although this guide discusses the implementation details for the supported languages, it does not list the syntax and definition of the elements of each language. This guide does not attempt to teach programmers how to write an application, nor does it attempt to teach C programming concepts.

Audience

The audience for this manual is the application programmer or system engineer who is already familiar with C programming and the programming development tools provided with the ULTRIX system.

Organization

This guide contains the following:

Chapter 1	Introduction to the Compiler System Provides an overview of each component of this compiler system and lists the options provided by each compiler driver.
Chapter 2	Storage Mapping (RISC) Describes how the compiler groups C and Pascal structures in storage for the RISC architecture.
Chapter 3	Storage Mapping (VAX) Describes how the compiler groups C structures in storage for the VAX architecture.
Chapter 4	Language Interfaces (RISC) Describes the coding interfaces between C and Pascal and provides information for calling and passing arguments between those languages for the RISC architecture.
Chapter 5	Language Interfaces (VAX) Describes the coding interfaces between C and Pascal and provides information for calling and passing arguments between those languages for the VAX architecture.

Chapter 6 Improving Program Performance (RISC)

Describes the profiling and optimization facilities that are available as part of the ULTRIX compiler system for the RISC architecture and that can be used to increase the efficiency of your programs.

Chapter 7 Improving Program Performance (VAX)

Describes the profiling and optimization facilities that are available as part of the ULTRIX compiler system for the VAX architecture and that can be used to increase the efficiency of your programs.

Chapter 8 Debugging

Describes the debugging tools that are available as part of the ULTRIX programming environment.

Chapter 9 Programming in a POSIX Environment

Describes the ULTRIX programming environment that lets you write programs that conform to specific standards.

Chapter 10 Security Guidelines for Programmers

Provides security guidelines for designing and writing programs.

Chapter 11 System Calls and Library Routines with Security Implications

Discusses the ULTRIX system calls and library routines that have security implications for programmers.

Appendix A C Implementation

Describes the extensions and modifications that are supported by this `cc` compiler and that differ from other C implementations.

New and Changed Information

This version of the guide contains information applicable to the VAX architecture in addition to updated information applicable to the RISC architecture. The chapter on debugging has been considerably expanded.

The guide contains a new chapter that describes how to program in a POSIX environment. It also contains two new chapters on security considerations for programmers.

Related Documents

ULTRIX Reference Pages

Contains many of the reference pages for the commands and tools that are described in this manual.

See the User's Guides for the individual programming languages for descriptions of each language. See the *Guide to Developing International Software* if your are writing programs for an international environment.

Conventions

>>> CPU*nn*>>	The console subsystem prompt is two right angle brackets on RISC systems, or three right angle brackets on VAX systems. On a system with more than one central processing unit (CPU), the prompt displays two numbers: the number of the CPU, and the number of the processor slot containing the board for that CPU.
user input	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in interactive examples to indicate system output and also in code examples and other screen displays. In text, this typeface is used to indicate the exact name of a command, option, partition, pathname, directory, or file.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
`cat(1)`	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat(1)` indicates that you can find the material on the `cat` command in Section 1 of the reference pages.
Mbyte	Throughout the text, the abbreviation Mbyte is used for megabyte. One megabyte equals 1,048,576 bytes.

Introduction to the Compiler System 1

This chapter describes the components of the compiler system and how to use them. The components are:

- Compiler system drivers
- Link editor
- Object file tools
- Archive libraries

1.1 Compiler System Drivers

The compiler system drivers are the programs that invoke the following compiler phases:

- The macro preprocessor (`cpp`)
- The compilers (`cc`, `f77`, `fort`, and `pc`)
- The assembler (`as`)
- The link editor (`ld`)

Table 1-1 shows the compilers for the RISC and VAX architectures and the type of availability for each.

Table 1-1: Compilers Available for RISC and VAX Processors

Compiler	Included with ULTRIX Kit for:	Layered Product for:
`as`	RISC, VAX	
`cc`	RISC, VAX	
`f77`		RISC
`fort`		VAX
`pc`	VAX	RISC

A separate driver exists for each language. This section provides an overview of each driver.

1.1.1 Driver Commands

Table 1-2 lists the languages available and the commands that invoke their respective drivers.

Table 1-2: Driver Commands for Specific Languages

Language	Driver Command
Assembler	`as`
FORTRAN 77 (RISC)	`f77`
FORTRAN (VAX)	`fort`
C	`cc`
Pascal	`pc`

The `cc`, `pc`, `f77`, `fort`, and `as` commands invoke the drivers that compile, optimize, assemble, and link edit your programs. Each command knows the appropriate libraries associated with the main program and passes only those libraries to the link editor.

1.1.2 RISC Driver Input and Output Files

Most drivers recognize the contents of an input file by its suffix. Table 1-3 lists the valid suffixes for languages available for the RISC architecture.

Table 1-3: RISC Driver Input and Output File Suffixes

Suffix	Description
.a	Object library.
.b	ucode object library.
.c	C source file.
.e	`efl` source file.
.f	FORTRAN 77 source file.
.i	Source is assumed to be that of the processing driver. For example: pc –c *source*.i In this case, *source*.i is assumed to contain Pascal source (`pc`).
.o	Object file.
.p	Pascal source file.
.r	`ratfor` source file.
.s	Assembly source file.
.u	ucode object file.

The `as` assembly driver assumes that any file, regardless of the suffix, contains assembly language statements and accepts only one input source file.

1.1.3 VAX Driver Input and Output Files

Most drivers recognize the contents of an input file by its suffix. Table 1-4 lists the valid suffixes for languages available for the VAX architecture.

Table 1-4: VAX Driver Input and Output File Suffixes

Suffix	Description
.a	Object library.
.c	C source file.
.e	`efl` source file.
.f	`fort` source file.
.o	Object file.
.p	Pascal source file.
.r	`ratfor` source file.
.s	Assembly source file.

The `as` assembly driver assumes that any file, regardless of the suffix, contains assembly language statements and accepts only one input source file.

1.1.4 Compiler System Components

When you compile a program, you usually select one or more options that affect a variety of program development functions, such as debugging, optimization, and profiling facilities, as well as the names assigned to output files.

Figure 1-1 illustrates the relationship between the major components of the compiler system and their primary inputs and outputs for the RISC driver.

Figure 1-1: Compiler Phases Used by the RISC Driver

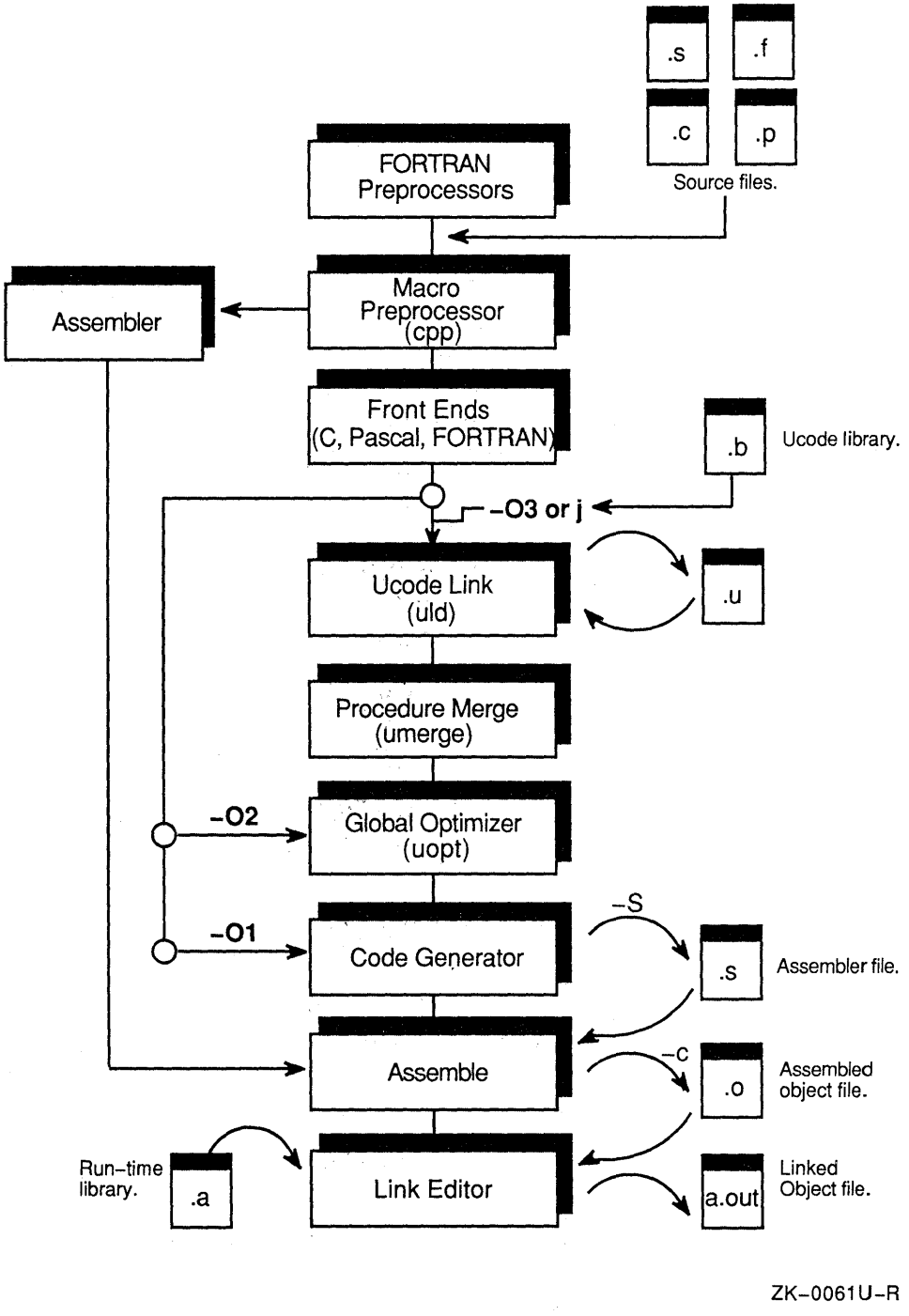
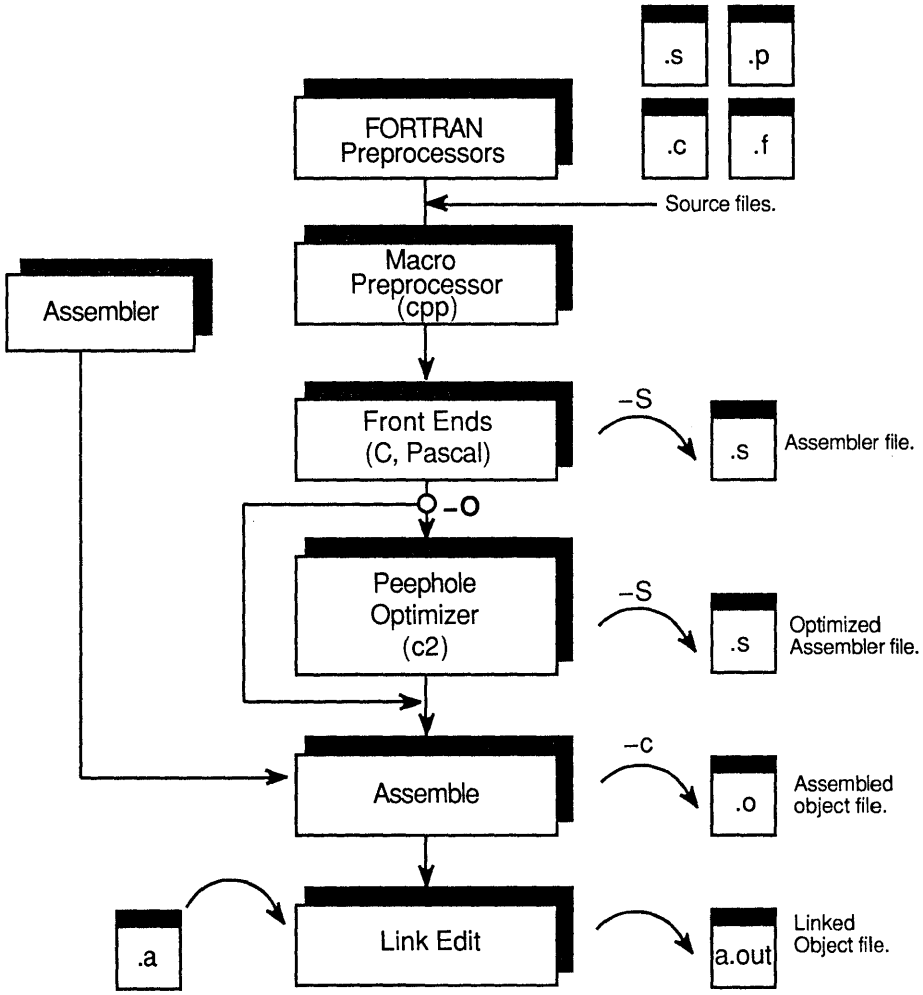


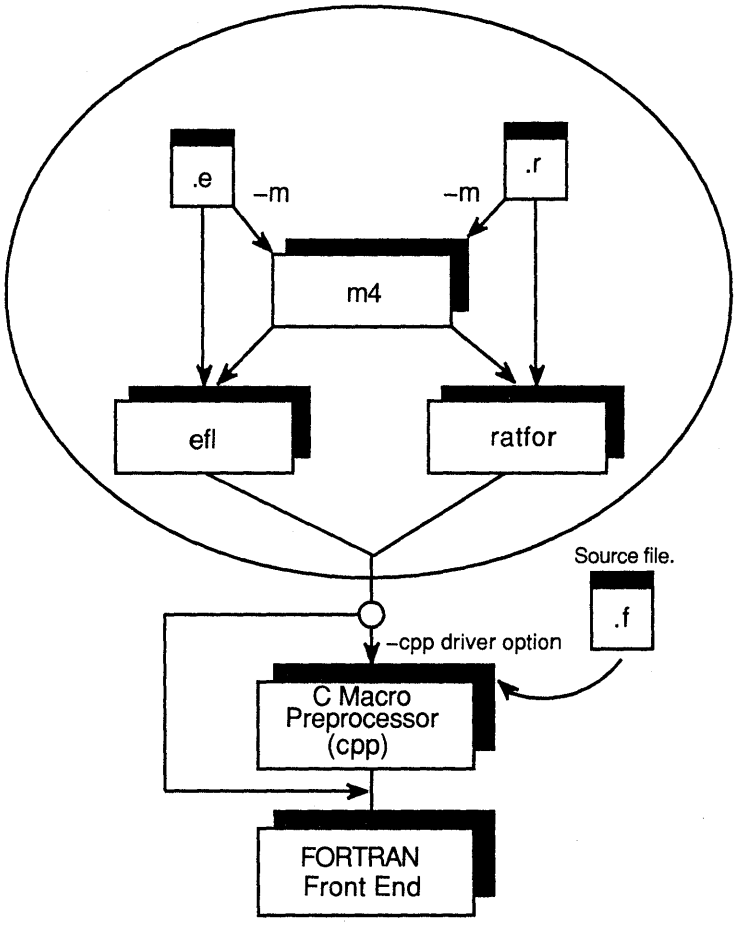
Figure 1-2 illustrates the relationship between the major components of the compiler system and their primary inputs and outputs for the VAX driver.

Figure 1-2: Compiler Phases Used by the VAX Driver



Note that FORTRAN uses preprocessors that the other languages do not use. Figure 1-3 illustrates the relationship of the FORTRAN preprocessors.

Figure 1-3: The FORTRAN Preprocessors



ZK-0062U-R

Some options have defaults. For example, the default name for object files is:

filename.o

The specified *filename* is the base name of the source file. The default name for executable program objects is `a.out`.

The following example shows compilation of two C source files, `foo.c` and `bar.c`, that generates object files with default names and a default executable program object. The following command invokes the compiler:

```
% cc foo.c bar.c
```

Following the flow illustrated in Figures 1-1 and 1-2, the C compiler compiles the source files (`foo.c` and `bar.c`), creates their respective object modules `foo.o` and `bar.o`, and their single executable program `a.out`.

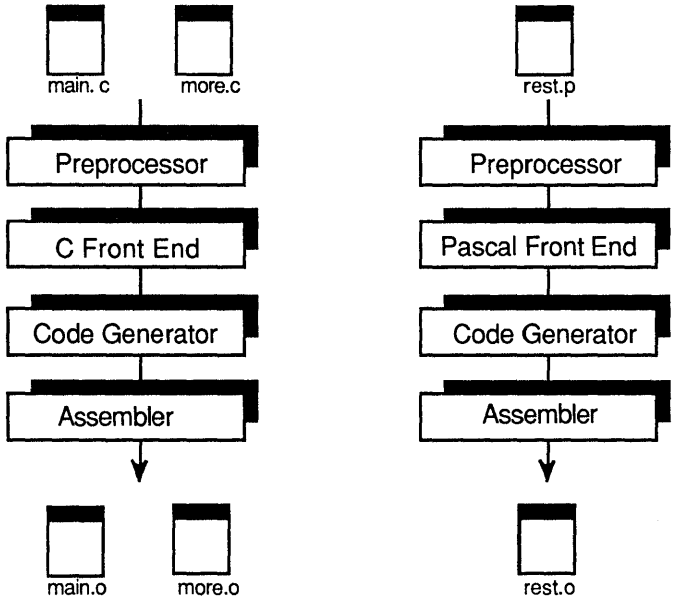
1.1.5 Compiling Multilanguage Programs

When the source language of the main program differs from that of a subprogram, you should compile each program module separately with the appropriate driver and then link them in a separate step. You can create objects suitable for link editing by specifying the `-c` option, which stops the driver immediately after the assembler phase. For example:

```
% cc -c main.c more.c
% pc -c rest.p
```

Figure 1-4 illustrates the compilation control flow for these commands.

Figure 1-4: Compiling Multilanguage Programs



ZK-0063U-R

1.1.6 Linking Objects

You can use a driver command to link-edit separate objects into one executable program. The driver recognizes the `.o` suffix as the name of a file that contains object code suitable for link editing and immediately invokes the link editor. You could link edit the objects created in the last example using the `pc` Pascal driver, as follows:

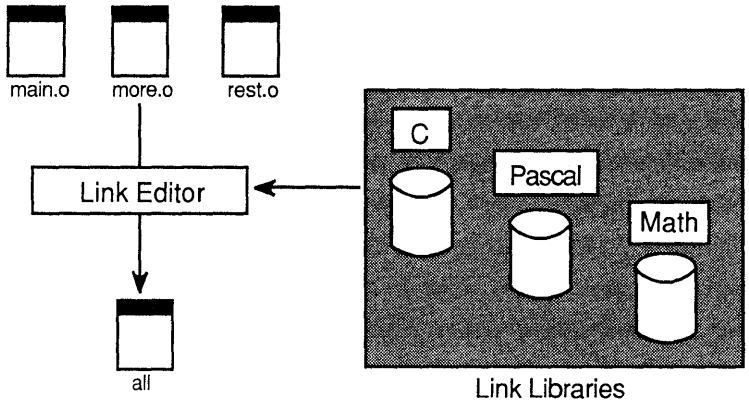
```
% pc -o all main.o more.o rest.o
```

This command produces the executable program object of the specified name, `all`. You could achieve the same results using the `cc` C driver, as follows:

```
% cc -o all main.o more.o rest.o -lp -lm
```

Figure 1-5 illustrates the control flow for the `pc` and `cc` commands used in these examples.

Figure 1-5: pc and cc Driver Control Flow



Note that to link the appropriate libraries with the `cc` driver, you must specify two additional options that the `pc` driver uses by default: the `-lp` option, which specifies the Pascal link library, and the `-lm` option, which specifies the math link library. Both `pc` and `cc` use the C link library by default.

For information on the link editor and on specifying link libraries, see Section 1.2.

For information about the default libraries used by each RISC driver, see `cc`(1) and `as`(1) in the *ULTRIX Reference Pages* and `f77`(1) and `pc`(1) in the Reference Pages for the FORTRAN and Pascal layered products.

For information about the default libraries used by each VAX driver, see `cc`(1), `pc`(1), and `as`(1) in the *ULTRIX Reference Pages* and `fort`(1) in the Reference Pages for the FORTRAN layered product.

1.1.7 Including Common Definition Files

When you write programs, you often have common definition files that you share among a program's modules. These files usually define known constants or the parameters for system calls (for example, the files that define the object file formats). Definition files, called `#include` or header files in the C programming language, let you share common information between files in a program. These header files typically have a `.h` suffix.

Each of the supported languages handles these files in the same way, and you specify these files in your program's source code.

If you intend to debug your program using the `dbx` debugger, do not place executable code in an include file. The debugger recognizes an include file as one line of source code; none of the source lines in the file appears during the debugging session.

To specify an include file in your program, you can use one of two methods. Whichever method you use, you should list all your include files in column 1 of your source file, as follows:

```
#include "file1"
```

```
#include "file2"
#include <file3>
#include <file4>
```

Each file name listed in this manner indicates the name of the include file. Because the names of the first two include files are in double quotation marks, the C macro preprocessor searches for them in the current directory and the default directory, `/usr/include`, in that order. Because the names of the next two include files are enclosed in angle brackets, the C macro preprocessor searches for them only in the default directory, `/usr/include`.

1.1.8 Setting Up Shareable Include Files in Programs (RISC Only)

For the RISC architecture, C, Pascal, FORTRAN 77, and assembly code can reside in the same `include` files and then can be conditionally included in programs as required. To set up a shareable include file, create a `.h` file and conditionalize the respective code as follows:

```
#ifdef LANGUAGE_C
      .
      .
      .
#endif
#ifdef LANGUAGE_PASCAL
      .
      .
      .
#endif
#ifdef LANGUAGE_FORTRAN
      .
      .
      .
#endif
#ifdef LANGUAGE_ASSEMBLY
      .
      .
      .
#endif
```

1.2 Link Editor

The link editor (`ld`) combines one or more object files (in the order specified) into one program object file, performing relocation, external symbol resolutions, and all other processing required to make object files ready for execution. Unless you specify otherwise, the link editor names the program object file `a.out`. You can execute the program object file or use it as input for another link editor operation.

The link editor supports all the standard command line features of other UNIX system link editors (except System V command language files, which contain a description of a load module).

For further information about the link editor, see `ld`(1) in the *ULTRIX Reference Pages*.

1.2.1 Running the Link Editor

To execute the link editor, use the following syntax:

ld *options object...*

Note that the `as` assembler does not automatically invoke the link editor. To link-edit a program written in assembly language, do either of the following:

- Assemble and link-edit by using one of the other drivers (for example, `cc`). The `.s` suffix of the assembly language source file automatically causes the driver to invoke the assembler procedures.
- Assemble by using `as`; then link-edit the resulting object file by using `ld`.

For further information about the options and libraries that affect the link editing process, see `ld`(1) in the *ULTRIX Reference Pages.*

1.2.2 Specifying Libraries

If you compile multilanguage programs, be sure to explicitly load any required run-time libraries. For example, if you write your main program in C and some procedures in Pascal, you must explicitly load the `libp.a` Pascal library and the `libm.a` main library by specifying the `-lp` and `-lm` options.

To find the Pascal library, `ld` replaces the `-l` with `lib` and adds the `.a` suffix. Then, it searches the `/lib`, `/usr/lib`, and `/usr/local/lib` directories for this library. For a list of the libraries that each language uses, see `cc`(1) in the *ULTRIX Reference Pages* and `f77`(1) and `pc`(1) in the Reference Pages for the FORTRAN and Pascal layered products.

You may need to specify libraries when you use UNIX system packages that are not part of a particular language. Most of the reference pages for these packages list the required libraries. For example, the plotting subroutines require the libraries listed in the `plot` reference page.

To specify a library created with the archiver, include the pathname of the library as part of the command syntax specified. For example, the following specifies that `libfft.a` is to be included along with the Pascal library:

```
% cc main.o more.o rest.o libfft.a -lp
```

The link editor searches libraries in the order you specify. Therefore, if you have a library (for example, `libfft.a`) that uses data or procedures from the Pascal library, you must specify it on the command line before you specify the Pascal library.

1.3 Object File Information

This section discusses the following commands that provide information on object files:

`file`	Provides descriptive information on the general properties of the specified file (for example, the programming language used).
`nm`	Lists symbol table information.

`odump`	Lists the contents (including the symbol table and header information) of an object file.
`size`	Prints the size of the code and data sections.

1.3.1 Determining a File's Type

The `file` command lists the properties of program source, text, object, and other files. Note that it often erroneously recognizes command files as C programs, and it does not recognize Pascal or LISP programs. To execute the `file` command, use the following syntax:

file *file...*

For further information, see `file`(1) in the *ULTRIX Reference Pages.*

1.3.2 Listing Symbol Table Information

The `nm` command prints symbol table information for object files and archive files. To execute the `nm` command, use the following syntax:

nm *options file...*

If you do not specify a file name, `nm` uses the default output file, `a.out`.

For further information, see `nm`(1) in the *ULTRIX Reference Pages.*

1.3.3 Dumping Selected Parts of Files (RISC Only)

For the RISC architecture, the `odump` command lists headers, tables, and other selected parts of an object or archive file. To execute the `odump` command, use the following syntax:

odump *options file...*

For further information, see `odump`(1) in the *ULTRIX Reference Pages.*

1.3.4 Determining a File's Section Sizes

The `size` command prints information about the code and data sections of the specified object or archive files. To execute the `size` command, use the following syntax:

size *options* [*file...*]

If you do not specify a file on the command line, `size` uses the default file, `a.out`.

For further information, see `size`(1) in the *ULTRIX Reference Pages.*

1.4 Archive Libraries

An archive library is a file that contains one or more routines in object (`.o`) file format. Here, the term object refers to an `.o` file that is part of an archive library file. When a program calls an object not explicitly included in the program, the link editor looks for that object in an archive library, then loads only that object (not the whole library), and links it with the calling program.

The `ar` archiver creates and maintains archive libraries by performing the following tasks:

- Copies new objects into the archive library.
- Replaces existing objects in the library.
- Moves objects within the library.
- Copies individual objects from the library into individual object files.

To execute the `ar` command, use the following syntax:

ar *keys* [*posname*] *afile name...*

The specified *posname* is the name of an object within an archive library. It specifies the relative placement (either before or after *posname*) of an object that is to be copied into the library or moved within the library.

For further information, see `ar`(1) in the *ULTRIX Reference Pages.*

Storage Mapping (RISC) 2

This chapter describes the alignment, size, and value ranges for C and Pascal structures and how the compiler groups these records in storage for the RISC architecture.

2.1 C Storage Mapping

The following sections describe how the compiler maps C variables into storage and discusses the following:

- Alignment, size, and value ranges
- C arrays, structures, and unions
- Storage classes

2.1.1 C Alignment, Size, and Value Ranges

Table 2-1 describes how the C compiler implements size, alignment, and value ranges for each data type for the RISC architecture.

Table 2-1: C Data Type Size, Alignment, and Value Ranges (RISC)

Type	Size	Alignment	Signed	Unsigned
int	32 bits	*Word*[1]	-2^{31} to $2^{31}-1$	0 to $2^{32}-1$
long	32 bits	*Word*[1]	-2^{31} to $2^{31}-1$	0 to $2^{32}-1$
enum	32 bits	*Word*[1]	-2^{31} to $2^{31}-1$	
short	16 bits	*Halfword*[2]	–32,768 to 32,767	0 to 65,535
char[3]	8 bits	Byte	–128 to 127	0 to 255
float[4]	32 bits	*Word*[1]	See next page	
double[5]	64 bits	*Doubleword*[6]	See next page	
pointer	32 bits	*Word*[1]	0 to $2^{32}-1$	

1. Byte boundary divisible by 4.
2. Byte boundary divisible by 2.
3. Unless the unsigned attribute is used, char is assumed to be signed.
4. IEEE single precision.
5. IEEE double precision.
6. Byte boundary divisible by 8.

The approximate valid ranges for the data types float and double for RISC processors are:

Type	Maximum Value	Minimum Value Normalized	Minimum Value Denormalized
float	$3.40282356 * 10^{38}$	$1.17549429 * 10^{-38}$	$1.40129846 * 10^{-46}$
double	$1.7976931348623158 * 10^{308}$	$2.2250738585072012 * 10^{-308}$	$4.9406564584124654 * 10^{-324}$

The `limits.h` and `float.h` header files, which are usually found in `/usr/include`, contain C macros that define minimum and maximum values for the various data types. For information about the macro names and values, see the appropriate header file.

2.1.2 C Arrays, Structures, and Unions

An array has the same boundary requirements as the data type specified for the array. The size of an array is the size of the data type multiplied by the number of elements. For example:

```
double x[2][3]
```

The size of the resulting array would be 48 (that is, 2*3*8, where 8 is the size in bytes of the double floating-point type).

Each member of a structure begins at an offset from the structure base. The offset corresponds to the order in which a member is declared; the first member is at offset 0.

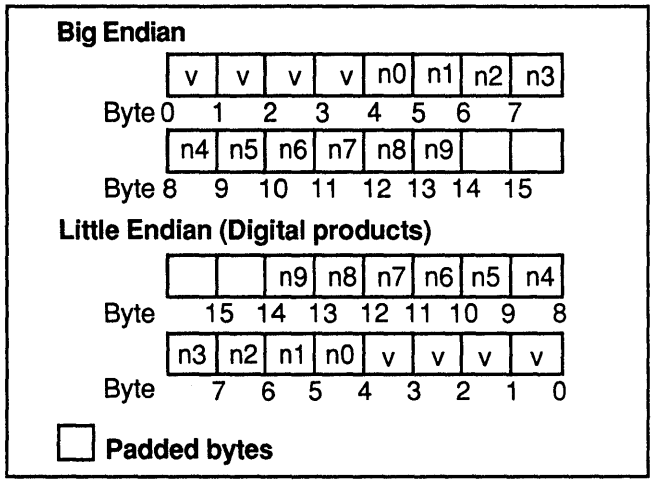
The size of a structure in the object file is the size of its combined members plus padding added, where necessary, by the compiler. The following rules apply to structures:

- Structures must align on the same boundary as that required by the member with the most restrictive boundary requirement. The boundary requirements, by increasing degree of restrictiveness, are byte, halfword, word, and doubleword.
- The compiler ends the structure on the same alignment boundary on which it begins. For example, if a structure begins on an even-byte boundary, it also ends on an even-byte boundary.

For example:

```
struct S {
     int v;
     char n[10];
}
```

The following figure illustrates how this structure would exist when mapped out in storage:



ZK-0065U-R

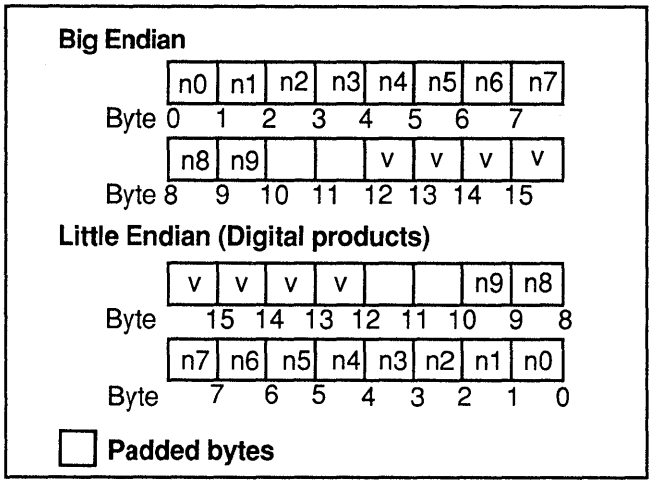
Even though the byte count defined by the `int v` and `char n` components is 14, the length of the structure is 16 bytes. Because `int` has a stricter boundary requirement (word boundary) than char (byte boundary), the structure must end on a word boundary (a byte offset divisible by 4). Therefore, the compiler adds two bytes of padding to meet this requirement.

An array of data structures illustrates the reason for this requirement. For example, if the structure in the previous figure were the element-type of an array, some of the `int v` components would not be aligned properly without the 2-byte pad.

Alignment requirements may cause padding to appear in the middle of a structure. For example:

```
struct S {
     char n[10];
     int v;
}
```

The following figure illustrates how this structure would exist when mapped out in storage:



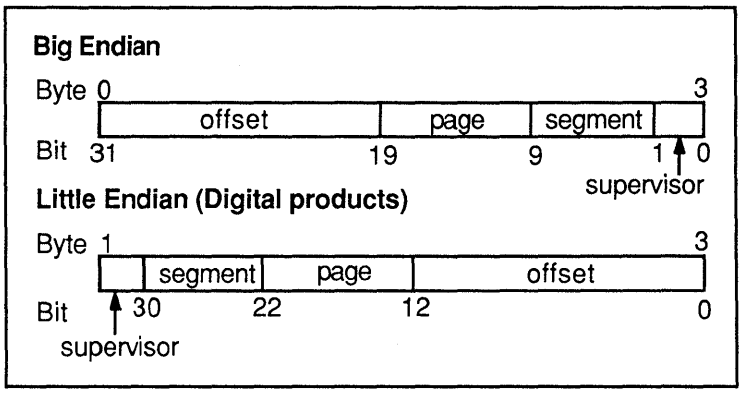
ZK-0066U-R

Note that the size of the structure remains 16 bytes, but two bytes of padding follow the `n` component to align `v` on a word boundary.

Bit fields are packed from the most-significant bit to least-significant bit in a word, cannot exceed 32 bits, and can be signed or unsigned. For example:

```
struct S {
     unsigned offset :12;
     unsigned page :10;
     unsigned segment :9;
     unsigned supervisor :1;
}virtual_address;
```

The following figure illustrates how this structure would exist when mapped out in storage:



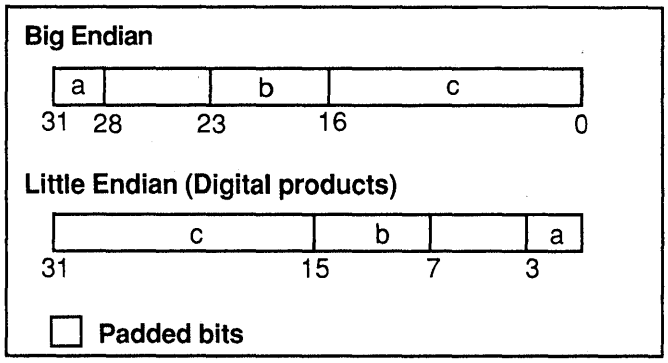
ZK-0067U-R

Note that the compiler moves the fields that overlap a word boundary to the next word.

The compiler aligns a nonbit field that follows a bit-field declaration to the next boundary appropriate for its type. For example:

```
struct S {
     unsigned a :3;
     char b;
     short c;
}x;
```

The following figure illustrates how this structure would exist when mapped out in storage:



ZK-0068U-R

Note that five bits of padding are added after `unsigned a` so that `char b` aligns on a byte boundary, as required.

A union must align on the same boundary as the member with the most restrictive boundary requirement. The boundary requirements, by increasing degree of restrictiveness, are: byte, halfword, word, and doubleword. For example, a union containing char, int, and double data types must align on a doubleword boundary, as is required by the double data type.

2.1.3 C Storage Classes

Table 2-2 lists the C storage classes for the RISC architecture.

Table 2-2: C Storage Classes (RISC)

Class	Description
auto	Indicates that storage is allocated at execution and exists only for the duration of that block activation.
static	Indicates that the compiler allocates storage, which remains fixed for the duration of the program, at compile time. Static variables reside in the program's bss section if they are not initialized; otherwise, they are placed in the data section.
register	Indicates that the compiler allocates variables with the register storage class to registers. For programs compiled with the `-O` option, the optimization phase of the compiler tries to assign all variables to registers, regardless of the storage class specified.
extern	Indicates that the variable refers to storage defined elsewhere in an external data definition. The compiler does not allocate storage to extern variable declarations; it uses the following logic in defining and referencing them:

Table 2-2: (continued)

Class	Description
	• *Extern is omitted* If an initializer is present, a definition for the symbol is emitted. Having two or more such definitions among all the files that form a program results in an error at link time or before. If no initializer is present, a common definition is emitted. Any number of common definitions of the same identifier can coexist. • *Extern is present* The compiler assumes that declaration refers to a name defined elsewhere. A declaration having an initializer is illegal. If a declared identifier is never used, the compiler does not issue an external reference to the linker.
volatile	Specified for variables that may be modified in ways unknown to the compiler. For example, volatile might be specified for an object corresponding to a memory mapped input/output port or an object accessed by an asynchronously interrupting function. Except for expression evaluation, no phase of the compiler optimizes any of the code dealing with volatile objects.

Note that if a pointer specified as volatile is assigned to another pointer without the volatile specification, the compiler treats the other pointer as nonvolatile. For example:

```
volatile int *i;
int *j;
    .
    .
    .
(volatile*)j = i;
```

The compiler treats `j` as a nonvolatile pointer and the object it points to as nonvolatile (the compiler may optimize it). Note that the `-volatile` compiler option causes all objects to be compiled as volatile.

2.2 Pascal Storage Mapping

The following sections describe how the compiler maps Pascal variables into storage and discuss the following:

- Alignment, size, and value ranges
- Pascal arrays and records

2.2.1 Pascal Alignment, Size, and Value Ranges

This section describes how the Pascal compiler implements size, alignment, and value ranges for the various data types. Table 2-3 describes the value ranges for the Pascal scalar types.

Table 2-3: Value Ranges for Pascal Scalar Types (RISC)

Scalar Types	Value Ranges
boolean	0 or 1
char	0 to 127
integer, integer32	-2^{31} to $2^{31}-1$
integer16	–32768 to 32767
cardinal	0 to $2^{32}-1$
real	See Note.
double	See Note.

Note

The approximate valid ranges for the data types real and double for RISC processors are:

Type	Maximum Value	Minimum Value Normalized	Minimum Value Denormalized
real	$3.40282356 * 10^{38}$	$1.17549429 * 10^{-38}$	$1.40129846 * 10^{-46}$
double	$1.7976931348623158 * 10^{308}$	$2.2250738585072012 * 10^{-308}$	$4.9406564584124654 * 10^{-324}$

Note that the enumerated types with n elements are treated the same as the integer subrange 0...n–1.

Table 2-4 describes the size and alignment of Pascal packed arrays.

Table 2-4: Size and Alignment of Pascal Packed Arrays (RISC)

Scalar Type	Size (Bits)	Alignment
boolean	8	Byte
char	8	Byte
integer, integer32	32	Word
integer16	16	Halfword
cardinal	32	Word
pointer	32	Word
file	32	Word
real	32	Word
double	64	Doubleword
0 .. 1 or –1 .. 0	1	Bit
0 .. 3 or –2 .. 1	2	2-bit
0 .. 15 or –8 .. 7	4	4-bit
0 .. 255 or –128 .. 127	8	Byte

Table 2-4: (continued)

Scalar Type	Size (Bits)	Alignment
0 .. 65535 or –32768 .. 32767	16	Halfword
0 .. 2^{32} –1 or -2^{31} .. -2^{31} –1	32	Word
set of char set of char subrange	128	Word
set of a to b		See Note

Note

The `set of a to b` is aligned on an *n*-bit boundary where *n* is computed as follows:

```
n = [log (size)]
```

For example, the set of `0 to 2` has a size of 3 bits and aligns on a 4-bit boundary.

(The notation [*x*] indicates the ceiling of *x*, that is, the smallest integer not less than *x*.)

The number of bits for `set of a to b` is calculated using the following formula:

```
if b-[a/32] * 32 + 1 <= 32 then
size = b-[a/32] * 32 + 1 bits
   else
          size =([b/32]-[a/32] + 1)*32 bits
```

Table 2-5 describes the size and alignment of Pascal unpacked records or arrays (variables or fields).

Table 2-5: Size and Alignment of Pascal Unpacked Records or Arrays (RISC)

Scalar Type	Size (Bits)	Alignment
boolean	8	Byte
char	8	Byte
integer, integer 32	32	Word
integer16	16	Halfword
cardinal	32	Word
pointer	32	Word
file	32	Word
real	32	Word
double	64	Doubleword

Table 2-5: (continued)

Scalar Type	Size (Bits)	Alignment
0 .. 255 or –128 .. 127	8	Byte
0 .. 65535 or –32768 .. 32767	16	Halfword
0 .. $2^{32}-1$ or -2^{31} .. $-2^{31}-1$	32	Word
set of char set of char subrange	128	Word
set of a to b	See Note	Word

Note

The compiler uses the following formula for determining the size of the `set of a to b`:

```
size = [b/32] - [a/32] + 1 words
```

(The notation [x] indicates the floor of x, that is, the largest integer not greater than x.)

Table 2-6 describes the size and alignment of Pascal packed records.

Table 2-6: Size and Alignment of Pascal Packed Records (RISC)

Scalar Type	Size (Bits)	Alignment
boolean	1	Bit
char	8	Bit
integer, integer 32	32	Word
integer16	16	Halfword
cardinal	32	Word
pointer	32	Word
file	32	Word
real	32	Word
double	64	Doubleword
subrange of	See Note	Bit/Word

Note

The compiler uses the minimum number of bits possible in creating a subrange field in a packed record. It uses the following formula:

```
If a >= 0 then size = [log2(b+1)]bits
If a =< 0 then size = max([log2(b+1)], [log2 (-a)]) +1 bits
```

To avoid crossing a word boundary, the compiler moves data types

aligned to bit boundaries in a packed record to the next word. This implies that any subrange appearing in a packed record causes the whole record to be word-aligned. This allows the compiler to always load words when extracting any field in the record and also implies that a subrange within a packed record corresponds to a bitfield declared with the `long int` base type in C.

The `of` extension is available for users who want the subrange to correspond to a bitfield in C with a base type other than `long int`:

```
type
   test = packed record
     a: 0..127 of char;
     b: 0..7 of char;
   end;
```

This example specifies that the subranges have a character base type rather than an integer base type, which is the default. Therefore, the resulting record has byte alignment. The compiler then loads bytes when extracting any field in the record. In allocating the field, the compiler moves the field to the next byte whenever the field would overlap a byte boundary.

The user can also specify `integer` and `integer16` data type using the `of` extension. If the `of` clause is not present, `integer` base type is assumed.

The record definition shown in the `of` extension example corresponds to the following structure definition in C:

```
struct test2 {
   unsigned char a:7;
   unsigned char b:7;
} :s;
```

2.2.2 Pascal Arrays and Records

The compiler maps Pascal arrays and records into storage exactly as it maps C arrays and structures.

2.2.3 Rules for Set Sizes

The maximum number of elements permitted in a set ranges between 481 and 512. This variance is due to the way Pascal implements sets. For efficient accessing of set elements, Pascal expects the lower-bound of a set to be a multiple of 32. For example, if you specify the following:

```
set of a to b
```

If `a` is not a multiple of 32, Pascal adds elements to the set from `a` down to the next multiple of 32 less than `a`. For example, if you specify the following:

```
set of 5..31
```

Pascal would add internal padding elements 0..4. These padding elements are inaccessible to the program. This implementation sacrifices some space for a fast, consistent method of accessing set elements.

The padding elements required to pad the lower bound down to a multiple of 32 varies between 0 and 31 elements.

The following condition must be met for `set of a to b` to be a valid set in Pascal:

```
size=(b-32[a/32]+1)<=512
```

Table 2-7 shows several sample sets and whether or not they are valid in Pascal.

Table 2-7: Example Sets (RISC)

Specification	Lower	Upper	Set Size	Valid Size
set of 1 to 511	0[1]	511	512	Yes
set of 0 to 511	0	511	512	Yes
set of 1 to 512	0[1]	512	513	No
set of 31 to 512	0[1]	512	513	No
set of 32 to 512	32	512	481	Yes
set of 32 to 543	32	543	512	Yes

1. As adjusted by Pascal

Storage Mapping (VAX) 3

This chapter describes the alignment, size, and value ranges for C structures and how the compiler groups these records in storage for the VAX architecture.

3.1 C Storage Mapping

The following sections describe how the compiler maps C variables into storage and discusses the following:

- Alignment, size, and value ranges
- C arrays, structures, and unions
- Storage classes

3.1.1 C Alignment, Size, and Value Ranges

Table 3-1 describes how the C compiler implements size, alignment, and value ranges for each data type for the VAX architecture.

Table 3-1: C Data Type Size, Alignment, and Value Ranges (VAX)

Type	Size	Alignment	Signed	Unsigned
int	32 bits	*Word*[1]	-2^{31} to $2^{31}-1$	0 to $2^{32}-1$
long	32 bits	*Word*[1]	-2^{31} to $2^{31}-1$	0 to $2^{32}-1$
enum	32 bits	*Word*[1]	-2^{31} to $2^{31}-1$	
short	16 bits	*Halfword*[2]	-32,768 to 32,767	0 to 65,535
char[3]	8 bits	Byte	-128 to 127	0 to 255
float	32 bits	*Word*[1]	See next page	
double	64 bits	*Doubleword*[4]	See next page	
pointer	32 bits	*Word*[1]	0 to $2^{32}-1$	

1. Byte boundary divisible by 4.
2. Byte boundary divisible by 2.
3. Unless the unsigned attribute is used, char is assumed to be signed.
4. Byte boundary divisible by 8.

The approximate valid ranges for the data types float and double for VAX processors are:

Type	Maximum Value	Minimum Value
float	$1.7014118 * 10^{38}$	$2.9387359 * 10^{-39}$
double (D-float)	$1.701411834604692291 * 10^{38}$	$2.93873587705571880 * 10^{-39}$
double (G-float)	$8.988465674311579 * 10^{307}$	$5.562684646268003 5 * 10^{-309}$

The `limits.h` and `float.h` header files, which are usually found in `/usr/include`, contain C macros that define minimum and maximum values for the various data types. For information about the macro names and values, see the appropriate header file.

3.1.2 C Arrays, Structures, and Unions

An array has the same boundary requirements as the data type specified for the array. The size of an array is the size of the data type multiplied by the number of elements. For example:

```
double x[2][3]
```

The size of the resulting array would be 48 (that is, 2*3*8, where 8 is the size in bytes of the double floating-point type).

Each member of a structure begins at an offset from the structure base. The offset corresponds to the order in which a member is declared; the first member is at offset 0.

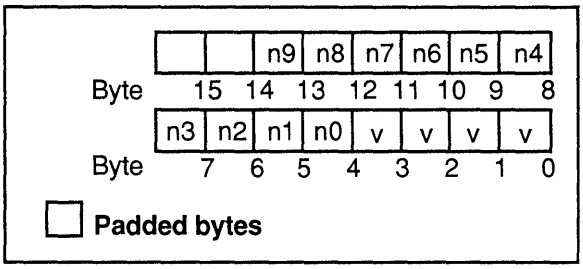
The size of a structure in the object file is the size of its combined members plus padding added, where necessary, by the compiler. The following rules apply to structures:

- Structures must align on the same boundary as that required by the member with the most restrictive boundary requirement. The boundary requirements, by increasing degree of restrictiveness, are byte, halfword, word, and doubleword.
- The compiler ends the structure on the same alignment boundary on which it begins. For example, if a structure begins on an even-byte boundary, it also ends on an even-byte boundary.

For example:

```
struct S {
     int v;
     char n[10];
}
```

The following figure illustrates how this structure would exist when mapped out in storage:



ZK-0065U1-R

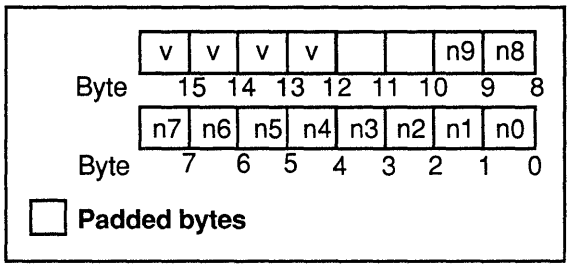
Even though the byte count defined by the `int v` and `char n` components is 14, the length of the structure is 16 bytes. Because `int` has a stricter boundary requirement (word boundary) than char (byte boundary), the structure must end on a word boundary (a byte offset divisible by 4). Therefore, the compiler adds two bytes of padding to meet this requirement.

An array of data structures illustrates the reason for this requirement. For example, if the structure in the previous figure were the element-type of an array, some of the `int v` components would not be aligned properly without the 2-byte pad.

Alignment requirements may cause padding to appear in the middle of a structure. For example:

```
struct S {
     char n[10];
     int v;
}
```

The following figure illustrates how this structure would exist when mapped out in storage:



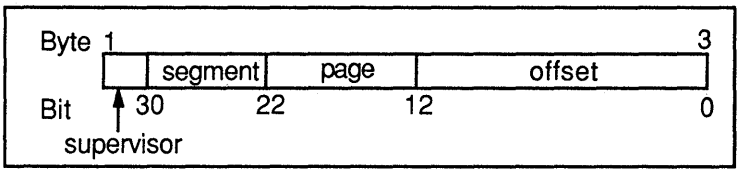
ZK-0066U1-R

Note that the size of the structure remains 16 bytes, but two bytes of padding follow the `n` component to align `v` on a word boundary.

Bit fields are packed from the most-significant bit to least-significant bit in a word, cannot exceed 32 bits, and can be signed or unsigned. For example:

```
struct S {
     unsigned offset :12;
     unsigned page :10;
     unsigned segment :9;
     unsigned supervisor :1;
}virtual_address;
```

The following figure illustrates how this structure would exist when mapped out in storage:



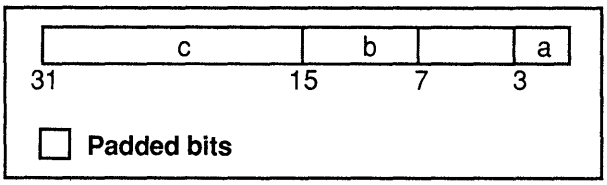
ZK-0067U1-R

Note that the compiler moves the fields that overlap a word boundary to the next word.

The compiler aligns a nonbit field that follows a bit-field declaration to the next boundary appropriate for its type. For example:

```
struct S {
     unsigned a :3;
     char b;
     short c;
}x;
```

The following figure illustrates how this structure would exist when mapped out in storage:



ZK-0068U1-R

Note that five bits of padding are added after `unsigned a` so that `char b` aligns on a byte boundary, as required.

A union must align on the same boundary as the member with the most restrictive boundary requirement. The boundary requirements, by increasing degree of restrictiveness, are: byte, halfword, word, and doubleword. For example, a union containing char, int, and double data types must align on a doubleword boundary, as is required by the double data type.

3.1.3 C Storage Classes

Table 3-2 lists the C storage classes for the VAX architecture:

Table 3-2: C Storage Classes (VAX)

Class	Description
auto	Indicates that storage is allocated at execution and exists only for the duration of that block activation.
static	Indicates that the compiler allocates storage, which remains fixed for the duration of the program, at compile time. Static variables reside in the program's bss section if they are not initialized; otherwise, they are placed in the data section.

Table 3-2: (continued)

Class	Description
register	Indicates that the compiler allocates variables with the register storage class to registers. For programs compiled with the -O option, the optimization phase of the compiler tries to assign all variables to registers, regardless of the storage class specified.
extern	Indicates that the variable refers to storage defined elsewhere in an external data definition. The compiler does not allocate storage to extern variable declarations; it uses the following logic in defining and referencing them: • *Extern is omitted* If an initializer is present, a definition for the symbol is emitted. Having two or more such definitions among all the files that form a program results in an error at link time or before. If no initializer is present, a common definition is emitted. Any number of common definitions of the same identifier can coexist. • *Extern is present* The compiler assumes that declaration refers to a name defined elsewhere. A declaration having an initializer is illegal. If a declared identifier is never used, the compiler does not issue an external reference to the linker.
const	The program is not permitted to alter the variable. The compiler may not detect all possible alterations (for example, by another function or by pointers).

Language Interfaces (RISC) 4

This chapter describes the coding interfaces between C and Pascal and gives rules and examples for calling and passing arguments between these languages for the RISC architecture.

This chapter discusses the following topics:

- General considerations
- Calling Pascal from C
- Calling C from Pascal

For detailed information on how the variables of the various languages appear in storage, see Chapter 2.

4.1 General Considerations

In general, calling C from Pascal and Pascal from C is fairly simple. Both Pascal and C allow only one main routine in a program, which can be written in either Pascal or C. Most data types in each language have natural counterparts in the other language. However, differences do exist in the following areas:

- Single-precision floating point
- Procedure and function parameters
- Pascal by-value arrays
- File variables
- Passing string data between C and Pascal
- Passing a variable number of arguments
- Type checking

4.1.1 Single-Precision Floating Point

In function calls, C automatically converts single-precision floating point values to double precision. By contrast, Pascal passes single-precision floating by-value arguments directly.

When passing double-precision values between C and Pascal routines, follow these guidelines:

- If possible, write the Pascal routine so that it receives and returns double-precision values.
- If the Pascal routine cannot receive a double-precision value, write a Pascal routine to accept double-precision values from C and then have that routine call the single-precision Pascal routine.

Passing single-precision values by reference between C and Pascal does not pose a problem.

4.1.2 Procedure and Function Parameters

C function variables and parameters consist of a single pointer to machine code. By contrast, Pascal procedure and function parameters consist of a pointer to machine code and a pointer to the stack frame of the lexical parent of the function. Such values can be declared as structures in C. To create such a structure, put the C function pointer in the first word and zero in the second. C functions cannot be nested and, thus, have no lexical parent. Therefore, the second word is irrelevant.

Note that you cannot call a C function with a function parameter from Pascal.

4.1.3 Pascal By-Value Arrays

C never passes arrays by value. In C, an array is actually a pointer. Therefore, passing an array actually passes its address, which corresponds to Pascal by-reference (VAR) array passing. In practice, this difference is not a serious problem because passing Pascal arrays by value is not very efficient. Therefore, most Pascal array parameters are VARs. When it is necessary to call a Pascal routine with a by-value array parameter from C, pass a C structure containing the corresponding array declaration.

4.1.4 File Variables

The Pascal text type and the C `stdio` package's declaration `FILE*` are compatible. However, Pascal passes file variables only by reference; a Pascal routine cannot pass a file variable by value to a C function. C functions that pass files to Pascal routines should, as with any reference parameter, pass the address of the `FILE*` variable.

4.1.5 Passing String Data Between C and Pascal

The C and Pascal languages handle strings differently. Pascal handles string data as fixed-length arrays of characters. String parameters are typically declared as follows:

```
VAR S: PACKED ARRAY[1..100] OF CHAR;
```

The upper bound (100 in this case) is large enough to handle most processing requirements efficiently. In passing an array, Pascal passes the entire array, as specified, and pads to the end of the array with spaces. Most C functions treat strings as pointers to a single character and use pointer arithmetic to step through the string. A null character (\0 in C) terminates a string in C. Therefore, when passing a string from Pascal to C, terminate the string with a null character (chr(0) in Pascal).

The following example shows a Pascal program that calls the `atoi` C function and passes the string `s`. Note that the program ensures that the string terminates with a null character.

```
type
  astrindex = 1 .. 20;
  astring = packed array [astrindex] of char;
  function atoi(var c:  astring): integer; external;
program ptest(output);
  var
    s: astring;
    i: astrindex;
```

```
begin
argv(1, s); { This predefined Pascal function
              is an extension }
writeln(output, s);
{ Guarantee that the string is null-terminated
  (but may eliminate the last character if the argument
  is too long). "lbound" and "hbound" are extensions. }
s[hbound(s)] := chr(0);
for i := lbound(s) to  hbound(s) do
  if s[i] = ' ' then
           begin
           s[i] := chr(0);
           break;
           end;
writeln(output, atoi(s));
end.
```

For more information on `atoi`, see `atof`(3) in the *ULTRIX Reference Pages.*

4.1.6 Passing a Variable Number of Arguments

C functions can be defined that take a variable number of arguments (for example, `printf` and its variants). Such functions can be called from Pascal, but they must be defined with a specific number of parameters in your Pascal program.

4.1.7 Type Checking

Pascal checks certain variables for errors at execution time; by contrast, C does not. For example, when a reference to an array exceeds its bounds in a Pascal program, the error is flagged (if run-time checks are not suppressed). You should not expect a C function to detect similar errors when you pass data to it from a Pascal program.

4.2 Calling Pascal from C

To call a Pascal function from C, write a C `extern` declaration to describe the return value type of the Pascal routine. Then, write the call with the return value type and argument types as required by the Pascal routine. The next sections discuss the following:

- C return values
- C-to-Pascal arguments

4.2.1 C Return Values

The following table provides guidelines for declaring a return value type:

Pascal Return Value Type	C Type Declaration
integer[1]	int
cardinal[2]	unsigned int
char	char
boolean	char
enumeration	unsigned or corresponding enum (signed in C)

Pascal Return Value Type	C Type Declaration
real	None
double	double
pointer type	Corresponding pointer type
record type	Corresponding structure or union type
array type	Corresponding array type

1. Applies also to subranges with lower bounds <0.
2. Applies also to subranges with lower bounds >=0.

To call a Pascal procedure from C, write a C extern declaration in the following form:

```
extern void name();
```

Then call it with actual arguments that have appropriate types.

4.2.2 C-to-Pascal Arguments

The following table lists the C argument types that match those expected by the called Pascal routine. Note that C does not permit declaration of the formal parameter types. Instead, it infers them from the types of the actual arguments passed.

Pascal Type Expected	C Type
integer	integer or char value -2^{31} .. $2^{31}-1$
cardinal	integer or char value 0 .. $2^{32}-1$
subrange	integer or char value in subrange
char	integer or unsigned char (0 to 127)
boolean	integer or char (0 or 1)
enumeration	integer or char (0 .. N–1)
real	None
double	float or double
procedure	struct *{void *p(); int *l}*
function	struct *{function-type *f(); int *l}*
pointer types	pointer type und <0. := lbound(s)
Reference parameter	Pointer to the appropriate type
record types	Structure or union type
by-reference array parameters	Corresponding array type
by-reference text	FILE**
by-value array parameters	Structure that contains the corresponding array

To pass a pointer to a function in a call from C to Pascal, you must pass a structure by value. The first word of the structure must contain the function pointer, and the second word must contain a zero. Pascal requires this format because it expects an environment specification in the second word.

The following is an example of a C function calling a Pascal function:

```
function bah(
     var f: text;
     i: integer
     ): double;
  begin
     .
     .
     .
  end {bah};
C declaration of bah:
extern double bah();
C call:
int i; double d;
FILE *f;
d = bah(&f, i);
```

The following is an example of a C function calling a Pascal procedure:

```
type
  int_array = array[1..100] of integer;
procedure zero (
    var a: int_array;
    n: integer
    ): integer;
  begin
     .
     .
     .
  end {zero};
C declaration:
extern void zero();
C call:
int a[100]; int n;
zero(a, n);
```

The following is an example of a C function that passes strings to a Pascal procedure, which then prints them. Note the following:

- The Pascal procedure must check for the null [chr(0)] character, which indicates the end of the string passed by the C routine.
- The Pascal procedure must not write to output; instead, it uses the stdout file-stream descriptor passed by the C routine.

```
C call:
/* Send the last command-line argument to Pascal routine */
#include <stdio.h>
main(argc, argv)
  int argc; char **argv;
  {
  FILE *temp = stdout;
  if (argc != 0)
    p_routine(&temp, argv[argc - 1]);
  }
Pascal procedure:
{ We assume the string passed to us by the routine
  will not exceed 100 bytes in length }
type
  astring = packed array [1..100] of char;
procedure p_routine(var f: text; var c: astring);
  var
    i: integer;
  begin
```

```
i := lbound(c);
while (i < hbound(c)) and (c[i] <> chr(0)) do
  begin
  write(f, c[i]);
  i := i + 1;
  end;
writeln(f);
end;
```

4.2.3 Calling C from Pascal

To call a C function from Pascal, write a Pascal declaration that describes the C function. Write a procedure declaration or, if the C function returns a value, write a function declaration. Write parameter and return value declarations that correspond to the C parameter types.

The following table describes the Pascal argument types that match those expected by the called C function:

Type Expected By C Function	Pascal Type
int[1]	integer
unsigned *int*[2]	cardinal
short[3]	integer (or –32768 .. 32767)
unsigned short	cardinal (or 0 .. 65535)
unsigned char	char
char[4]	integer (or –128 .. 127)
float	double
double	double
enum type	Corresponding enumeration type
FILE *	text (passed by reference – VAR)
FILE **	Corresponding pointer type or corresponding type passed by reference (VAR)
struct type	Corresponding record type
union type	Corresponding record type
array type	Corresponding array type passed by reference (VAR)

1. Same as types signed int, long, signed long, signed.
2. Same as types unsigned, unsigned long.
3. Same as type signed short.
4. Same as type signed char.

Note that a Pascal routine cannot pass a function pointer to a C function.

The following is an example of a Pascal program calling a C procedure:

```
C function:
void bah(i, f, s)
    int i;
    float f;
    char *s;
{
    .
    .
    .
}
```

Pascal declaration:

```
procedure bah(
    i: integer;
    f: double;
    var s: packed array[1..100] of char)
  external;
```

Pascal call:

```
str := "abc\0"
bah(i, 1.0, str)
```

The following is an example of a Pascal program calling a C function:

C function:

```
float humbug(f, x)
    FILE **f;
    struct scrooge *x;
{
     .
     .
     .
}
```

Pascal declaration:

```
type
  scrooge_ptr = ^scrooge;
function humbug(
    var f: text;
    x: scrooge_ptr
    ): double;
  external;
```

Pascal call:

```
x := humbug(input, sp);
```

The following is an example of a Pascal program calling a C function:

C function:

```
int sum(a, n)
    int a[];
    unsigned n;
{
     .
     .
     .
}
```

Pascal declaration:

```
type
  int_array = array[0..100] of integer;
function sum(
  var a: int_array;
  n: cardinal
  ): integer;
external;
avg := sum(samples,hbound(samples) +1) /
           (hbound(samples) +1);
```

Language Interfaces (VAX) 5

This chapter describes the coding interfaces between C and Pascal and gives rules and examples for calling and passing arguments between these languages for the VAX architecture.

This chapter discusses the following topics:

- General considerations
- Calling Pascal from C
- Calling C from Pascal

For detailed information on how the variables in C appear in storage, see Chapter 3.

5.1 General Considerations

In general, calling C from Pascal and Pascal from C is fairly simple. Both Pascal and C allow only one main routine in a program, which can be written in either Pascal or C. Most data types in each language have natural counterparts in the other language. However, differences do exist in the following areas:

5.1.1 Single-Precision Floating Point

In function calls, C automatically converts single-precision floating point values to double precision. By contrast, Pascal always uses double-precision and passes floating by-value arguments directly.

5.1.2 Passing String Data Between C and Pascal

The C and Pascal languages handle strings differently. Pascal handles string data as fixed-length arrays of characters. String parameters are typically declared as follows:

```
VAR S: PACKED ARRAY[1..100] OF CHAR;
```

The upper bound (100 in this case) is large enough to handle most processing requirements efficiently. In passing an array, Pascal passes the entire array, as specified, and pads to the end of the array with spaces. Most C functions treat strings as pointers to a single character and use pointer arithmetic to step through the string. A null character (\0 in C) terminates a string in C. Therefore, when passing a string from Pascal to C, terminate the string with a null character (chr(0) in Pascal).

The following example shows a Pascal program that calls the `atoi` C function and passes the string `s`. Note that the program ensures that the string terminates with a null character.

```
program example(output);
type
    examplestr = packed array [1..10] of char;
```

```
var
    s : examplestr;
    i : integer;
function atoi( var s : examplestr ) : integer; external;
begin
    s := '100';
    s[4] := chr(0);
    i := atoi(s);
    writeln(i);
end.
```

For more information on `atoi`, see `atof`(3) in the *ULTRIX Reference Pages*.

5.1.3 Passing a Variable Number of Arguments

C functions can be defined that take a variable number of arguments (for example, `printf` and its variants). Such functions can be called from Pascal, but they must be defined with a specific number of parameters in your Pascal program.

5.1.4 Type Checking

Pascal checks certain variables for errors at execution time; by contrast, C does not. For example, when a reference to an array exceeds its bounds in a Pascal program, the error is flagged (if run-time checks are not suppressed). You should not expect a C function to detect similar errors when you pass data to it from a Pascal program.

5.2 Calling Pascal from C

To call a Pascal function from C, write a C `extern` declaration to describe the return value type of the Pascal routine. Then, write the call with the return value type and argument types as required by the Pascal routine. The next sections discuss the following:

- C return values
- C-to-Pascal arguments

5.2.1 C Return Values

The following table provides guidelines for declaring a return value type:

Pascal Return Value Type	C Type Declaration
integer[1]	int
None	unsigned int
char	char
boolean	char
enumeration	unsigned or corresponding enum (signed in C)
real	double
pointer type	Corresponding pointer type

Pascal Return Value Type	C Type Declaration
record type	Corresponding structure or union type
array type	Corresponding array type

1. Applies also to subranges with lower bounds <0.

To call a Pascal procedure from C, write a C extern declaration in the following form:

```
extern void name();
```

Then call it with actual arguments that have appropriate types.

5.2.2 C-to-Pascal Arguments

The following table lists the C argument types that match those expected by the called Pascal routine. Note that C does not permit declaration of the formal parameter types. Instead, it infers them from the types of the actual arguments passed.

Pascal Type Expected	C Type
integer	integer or char value -2^{31} .. $2^{31}-1$
subrange	integer or char value in subrange
char	integer or unsigned char (0 to 127)
boolean	integer or char (0 or 1)
enumeration	integer or char (0 .. N–1)
real	double
pointer types	pointer type und <0. := lbound(s)
Reference parameter	Pointer to the appropriate type
record types	Structure or union type
by-reference array parameters	Corresponding array type
by-value array parameters	Structure that contains the corresponding array

The following is an example of a C function that passes strings to a Pascal procedure, which then prints them. Note the following:

- The Pascal procedure must check for the null [chr(0)] character, which indicates the end of the string passed by the C routine.
- The Pascal procedure must not write to output; instead, it uses the stdout file-stream descriptor passed by the C routine.

```
C call:
/* Send the last command-line argument to Pascal routine */
#include <stdio.h>
main(argc, argv)
```

```
  int argc; char **argv;
  {
  if (argc != 0)
    p_routine(argv[argc - 1]);
  }
```

Pascal procedure:

```
{ We assume the string passed to us by the routine
  will not exceed 100 bytes in length }
type
  astring = packed array [1..100] of char;
procedure p_routine(var c: astring);
  var
    i: integer;
  begin
  i := 1;
  while (i < 100) and (c[i] <> chr(0)) do
    begin
    write(c[i]);
    i := i + 1;
    end;
  writeln;
  end;
```

5.2.3 Calling C from Pascal

To call a C function from Pascal, write a Pascal declaration that describes the C function. Write a procedure declaration or, if the C function returns a value, write a function declaration. Write parameter and return value declarations that correspond to the C parameter types.

The following table describes the Pascal argument types that match those expected by the called C function:

Type Expected By C Function	Pascal Type
int[1]	integer
unsigned *int*[2]	None
short[3]	integer (or –32768 .. 32767)
unsigned short	0 .. 65535
unsigned char	char
char[4]	integer (or –128 .. 127)
double	real
enum type	Corresponding enumeration type
struct type	Corresponding record type
union type	Corresponding record type
array type	Corresponding array type passed by reference (VAR)

1. Same as types signed int, long, signed long, signed.
2. Same as types unsigned, unsigned long.
3. Same as type signed short.
4. Same as type signed char.

Note that a Pascal routine cannot pass a function pointer to a C function.

Improving Program Performance (RISC) 6

This chapter describes facilities that can help reduce the execution time of your programs on RISC processors. It discusses the following topics:

- Profiling code
- Optimizing code
- Controlling the size of global pointer data

The best way to produce efficient code is to follow good programming practices:

- Choose good algorithms and leave the details to the compiler.
- Avoid tailoring your work for any particular release or quirk of the compiler system.

6.1 Profiling Code

The profiler isolates those portions of your code where execution is concentrated and provides reports that indicate where you should devote your time and effort for coding improvements. This section describes the advantages of the profiler and how to use it.

In a typical program, execution time is confined to a relatively few sections of code, and it is profitable to concentrate on improving coding efficiency in only those sections. The compiler system provides the following profile information:

- Program counter (pc) sampling
- Invocation counting
- Basic block counting

The program counter highlights the execution time spent in various parts of the program. You can obtain pc sampling information by link-editing the desired source modules with the `-p` option and then executing the resulting program object, which generates profile data in raw format. Your program must exit normally for the profile data to be created.

Invocation counting gives the number of times each procedure in the program is invoked. Basic block counting measures the execution of basic blocks (a basic block is a sequence of instructions that is entered only at the beginning and which exits only at the end). This option provides statistics on individual lines.

You can obtain invocation counting and basic block counting information by using the `pixie` command, which uses your source program to create an equivalent program that contains additional code that counts the execution of each basic block. Executing `pixie` and the equivalent program generate the profile data in raw format.

For more information, see `pixie`(1) in the *ULTRIX Reference Pages*.

In addition, by using the `prof` command, you can create a formatted listing of the raw profile data. You can use this listing to determine if your program exercised all portions of your code, to determine where to correct inefficient code, or to determine where to substitute better algorithms or assembly code.

For further information, see `prof`(1) in the *ULTRIX Reference Pages*.

The following is an example of a pc sampling listing that was produced from a program compiled with the `-p` compiler option. The program was then executed. The `prof` command produced the listing from the raw profile data generated during execution. The output is sorted in descending order by the total time spent in each procedure; unexecuted procedures are excluded.

```
Each sample covers 8.00 byte(s) for 4.2% of 0.2400 seconds
%time  seconds   cum %  cum sec  procedure (file)
 25.0   0.0600    25.0     0.06  main (fixfont.p)
 16.7   0.0400    41.7     0.10  write_string (../textoutput.c)
 12.5   0.0300    54.2     0.13  write_char (../textoutput.c)
 12.5   0.0300    66.7     0.16  write_integer (../textoutput.c)
  8.3   0.0200    75.0     0.18  write (../write.s)
  4.2   0.0100    79.2     0.19  writeln (../textoutput.c)
  4.2   0.0100    83.3     0.20  write_chars (../textoutput.c)
  4.2   0.0100    87.5     0.21  read (../read.s)
  4.2   0.0100    91.7     0.22  open (../open.s)
  4.2   0.0100    95.8     0.23  rewrite (../rewrite.c)
  4.2   0.0100   100.0     0.24  eoln (../textinput.c)
```

The next three examples are `prof` command listings generated from raw data produced by the `pixie` command.

In the first example, the `prof -i` option was specified. The procedures are sorted in descending order by the number of calls. A question mark (?) in the #calls and line columns indicates that data is unavailable because part of the program was compiled without profiling.

```
called procedure  #calls  %calls  from line, calling procedure (file):
eoln                4017   81.51    37  main (pix.p)
                     453    9.19    35  main (pix.p)
                     428    8.69    19  main (pix.p)
                      30    0.61    17  main (pix.p)
write_char          4014   81.75    43  main (pix.p)
                     453    9.23    45  main (pix.p)
                     442    9.00    42  main (pix.p)
                       1    0.02    47  main (pix.p)
read_char           4017   90.37    36  main (pix.p)
                     428    9.63    18  main (pix.p)
write_chars         1382   60.40   160  write_string (../textoutput.c)
                     906   39.60   225  write_integer (../textoutput.c)
                       0    0.00   257  write_cardinal (../textoutput.c)
                       0    0.00   284  write_real(../textoutput.c)
                       0    0.00   286  write_real (../textoutput.c)
write_string         453   24.59    31  main (pix.p)
                     453   24.59    29  main (pix.p)
                     453   24.59    31  main (pix.p)
                     453   24.59    31  main (pix.p)
                      30    1.63    23  main (pix.p)
                       0    0.00   189  write_enum (../textoutput.c)
write_integer        453   50.00    31  main (pix.p)
                     453   50.00    31  main (pix.p)
eof                  453   93.40    45  main (pix.p)
                      30    6.19    23  main (pix.p)
                       1    0.21    28  main (pix.p)
                       1    0.21    14  main (pix.p)
writeln              453   93.60    29  main (pix.p)
```

```
                    30    6.20    23  main (pix.p)
                     1    0.21    47  main (pix.p)
readln             453   93.79    39  main (pix.p)
                    30    6.21    21  main (pix.p)
sbrk                 4   66.67   207  morecore (../malloc.c)
                     1   16.67   110  malloc (../malloc.c)
                     1   16.67   115  malloc (../malloc.c)
close                4  100.00   108  fclose (../flsbuf.c)
fflush               4  100.00   107  fclose (../flsbuf.c)
                     0    0.00    49  _filbuf (../filbuf.c)
```

In the second example, the `prof -p` option was specified. The output is sorted in descending order by the number of cycles executed in each procedure; unexecuted procedures are excluded.

```
 148137751 cycles
  cycles  %cycles  cum %    cycles  bytes procedure (file)
                             /call  /line
48071708    32.45  32.45        34     32 write_char (../textoutput.c)
42443503    28.65  61.10  42443503     26 main (fixfont.p)
26457936    17.86  78.96        30     44 eoln (../textinput.c)
20662326    13.95  92.91        23     27 read_char (../textinput.c)
 4307932     2.91  95.82        62      8 write_chars (../textoutput.c)
 3678408     2.48  98.30       133     14 write_integer (../textoutput.c)
 1573858     1.06  99.36        29     16 write_string (../textoutput.c)
  362700     0.24  99.61        26     67 readln (../textinput.c)
  279002     0.19  99.80        20     30 writeln (../textoutput.c)
  251152     0.17  99.97        19     44 eof (../textinput.c)
   30283     0.02  99.99        63     11 _flsbuf (../flsbuf.c)
   13391     0.01 100.00        60     13 _refill (../refill.c)
    2923     0.00 100.00         6      6 write (../write.s)
    1356     0.00 100.00         6      6 read (../read.s)
     735     0.00 100.00       368     11 morecore (../malloc.c)
     116     0.00 100.00        58     10 malloc (../malloc.c)
     105     0.00 100.00        15      9 pad (../textoutput.c)
      90     0.00 100.00        45     15 reset (../reset.c)
      82     0.00 100.00        82     13 fopen (../fopen.c)
      55     0.00 100.00        11      5 sbrk (../sbrk.s)
      35     0.00 100.00        35     15 rewrite (../rewrite.c)
      15     0.00 100.00         5      5 fstat (../fstat.s)
      13     0.00 100.00        13     12 isatty (../isatty.c)
      11     0.00 100.00        11     11 gtty (../gtty.c)
       6     0.00 100.00         6      5 ioctl (../simple.s)
       5     0.00 100.00         5      5 creat (../stringarg1.s)
       5     0.00 100.00         5      5 creat (../stringarg1.s)
```

In the third example, the `prof -l` option was specified. The output is grouped by procedure and sorted by the number of cycles executed per procedure. A question mark (?) indicates that data is unavailable because part of the program was compiled without profiling.

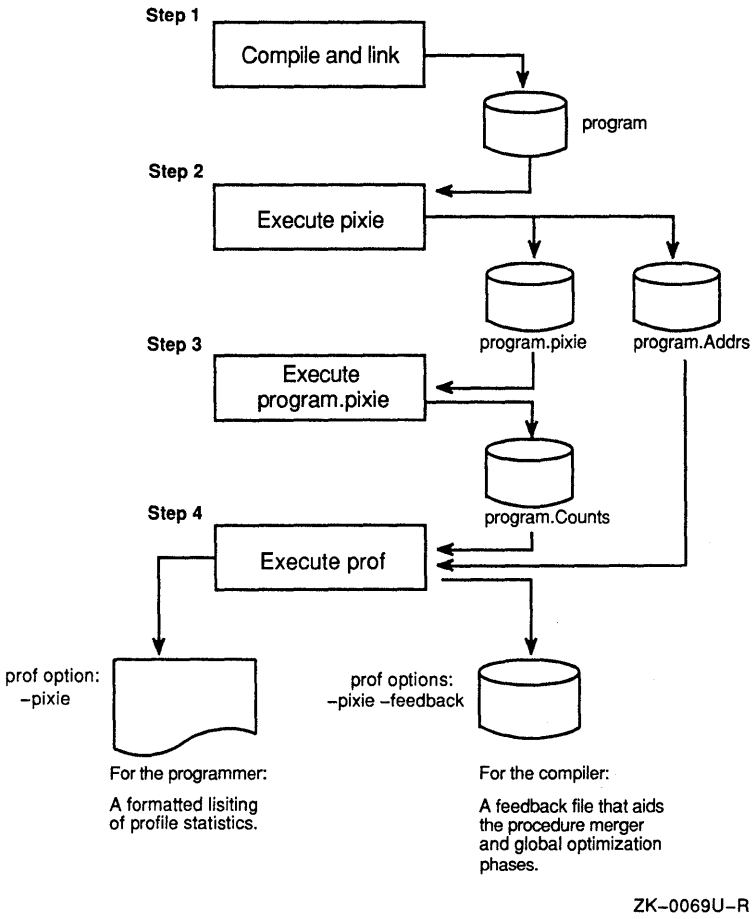
```
procedure (file)              line  bytes    cycles  %cycles
write_char (../textoutput.c)   105     20   7069725     4.77
                               106      8   2827890     1.91
                               111      8   2827890     1.91
                               106      4   1413945     0.95
                               112     16   1413945     0.95
                               113     72         0     0.00
                               115     12   4241835     2.86
                               116     64         0     0.00
                               117     28         0     0.00
                               120     88  28276478    19.09
main (fixfont.p)               11     60        15     0.00
                               12     32         8     0.00
```

```
                                    13      24          6      0.00
                                    14      24          6      0.00
                                    15       4          1      0.00
                                    16      40       8490      0.01
                                    17      24        166      0.00
                                    48      48         12      0.00
eoln (../textinput.c)               27      12    2736936      1.85
                                    28       4     912312      0.62
                                    31     116   22808688     15.40
read_char (../textinput.c)          58       8    1796724      1.21
                                    59      56    9881982      6.67
                                    60      28    5390172      3.64
                                    61      16    3593448      2.43
write_chars (../textoutput.c)       18      20     348150      0.24
                                    19       8     139260      0.09
                                    25      12     208890      0.14
                                    28       4        139      0.00
```

6.1.1 Basic Block Counting

Figure 6-1 illustrates the steps to follow in obtaining basic block counts.

Figure 6-1: Basic Block Counting



To obtain basic block counts, follow this procedure:

1. Compile and link-edit your program. Do not use the `-p` option. For example:

```
% cc -c myprog.c
% cc -o myprog myprog.o
```

2. Run the `pixie` profiling command on your compiled output. For example:

```
% pixie -o myprog.pixie myprog
```

   The `pixie` command creates an equivalent program containing additional code that counts the execution of each basic block. It also generates another output file (`myprog.Addrs`) that contains the address of each of the basic blocks. For more information, see `pixie`(1) in the *ULTRIX Reference Pages*.

3. Execute the `pixie`-generated output program, `myprog.pixie`, which in turn generates another output file (`myprog.Counts`), which contains the basic block counts.

4. Run the `prof` profile formatting command, which extracts and formats information from the secondary output files, `myprog.Addr` and `myprog.Counts`. For example:

```
% prof -pixie myprog myprog.Addrs myprog.Counts
```

   Note that specifying `myprog.Addrs` and `myprog.Counts` is optional; by default, `pixie` automatically searches the current directory for files with the specified program name and the `.Addrs` and `.Counts` suffixes.

   Also note that you can run the program several times, altering the input data, and create multiple profile data files. For further information, see Section 6.1.2.

You can include or exclude information on specific procedures within your program by using `prof` with the `-only` or `-exclude` option.

6.1.2 Averaging prof Results

A single run of a program may not produce the typical results you require. You can repeatedly run the version of your program created with the `pixie` command and vary the input with each run. Then, you can use each resulting `.Counts` file to produce a consolidated report. For example, the following would create four `.Counts` files from which one consolidated report can be generated.

1. Compile and link-edit your program. Do not use the `-p` option. For example:

```
% cc -c myprog.c
% cc -o myprog myprog.o
```

2. Run the `pixie` profiling command on the resulting program. For example:

```
% pixie -o myprog.pixie myprog
```

   This step produces the modified program `myprog.pixie` and the additional `myprog.Addrs` output file that is to be used later.

3. Run the profiled program as many times as needed for the different input. A `.Counts` file is generated each time that you run the profiled program. You should rename this file before executing the next sample run. For example:

```
% myprog.pixie < input1 > output1
% mv myprog.Counts myprog1.Counts
% myprog.pixie < input2 > output2
```

```
% mv myprog.Counts myprog2.Counts
% myprog.pixie < input3 > output3
% mv myprog.Counts myprog3.Counts
```

4. Create the consolidated report by including all of the generated `.Counts` files on the command line. For example:

```
% prof -pixie myprog myprog.Addrs myprog[123].Counts
```

The `prof` command takes an average of the basic block data in the named `.Counts` files to produce one consolidated profile report.

6.1.3 PC Sampling

To obtain pc sampling information, follow this procedure:

1. Compile and link-edit your program with the `-p` option. For example:

```
% cc -c myprog.c
% cc -p -o myprog myprog.o
```

   Note that you must specify the `-p` profiling option during the link-editing step to obtain pc sampling information.

2. Execute the profiled program. During its execution, profiling data is saved in the profile data file. The default is `mon.out`.

   You can run the program several times, alter the input data, and create multiple profile data files. For further information, see Section 6.1.4.

3. Run the `prof` profile formatter, which extracts information from each profile data file and formats it in an easily readable form. For example:

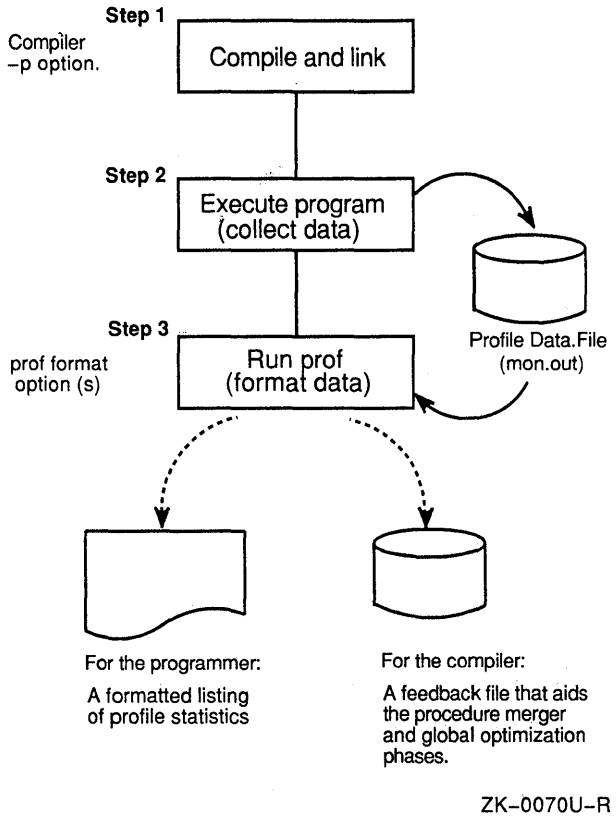
```
% prof -procedure myprog mon.out
```

You can include or exclude information on specific procedures within your program by using the `-only` or `-exclude` profiler option.

For further information, see `prof`(1) in the *ULTRIX Reference Pages*.

Figure 6-2 illustrates the procedure for obtaining pc sampling information.

Figure 6-2: PC Sampling



6.1.4 Creating Multiple Profile Data Files

When you run a program using pc sampling, raw data is collected and saved in the profile data file `mon.out`. To collect profile data in several files or to specify a different name for the profile data file, set the PROFDIR environment variable by using one of the following shell commands:

C Shell:

setenv PROFDIR *directory*

Bourne Shell:

PROFDIR = *directory*; export PROFDIR

Once you set your PROFDIR environment variable, the profiling results are then saved in *directory/pid.progname*, where *directory* is that specified by the PROFDIR environment variable, *pid* is the process ID of the executing program, and *progname* is the program's name as it appears in argv[0]. *pid* is the process ID of the executing program, and *progname* is the program's name as it appears in argv[0]. Note that you must create the PROFDIR directory before you run the program.

6.1.5 Running the prof Profiler

The `prof` profiler converts the raw profiling information into either a printed listing or an output file that can be used by the compiler. To execute the profiler, use the following syntax:

prof [*options*] [*file*] [*program*.Addrs *program*.Counts]

If you do not specify a profile data file, `prof` looks for `mon.out` in the current directory. If `mon.out` does not exist, `prof` looks for the profile data file in the directory specified by the PROFDIR environment variable.

If you do not specify a profile data file but you do specify the `-pixie` option, `prof` looks for the specified *program*.Addrs and *program*.Counts files and provides basic block count information if they are present. You can merge the data from multiple profile files into one new file.

For further information, see `prof`(1) in the *ULTRIX Reference Pages.*

6.2 Optimizing Code

The following sections provide an overview of the compiler optimization facilities and describes their benefits, the implications of optimizing and debugging, and the major optimizing techniques. They also give examples showing optimization techniques.

6.2.1 Overview of the Optimizer

The global optimizer improves the performance of object programs by transforming existing code into more efficient coding sequences. Although the same optimizer processes all compiler optimizations, it does distinguish between the various languages supported by the compiler system programs to take advantage of the different language semantics involved.

Most compilers perform certain code optimizations, although the extent to which they perform these optimizations varies widely. This compiler system performs more extensive optimizations compared with the average compiler available. These advanced optimizations are the results of the latest research into better and more powerful compiler techniques.

The compiler system performs both machine-independent and machine-dependent optimizations. Machines with RISC architectures provide a better target for machine-dependent optimizations, because the low-level instructions of RISC machines provide more optimization opportunities than the high-level instructions in other machines. Even optimizations that are machine independent have been found to be effective on machines with RISC architectures. Although most of the optimizations performed by the global optimizer are machine independent, they have been specifically tailored to this RISC environment.

The RISC architecture emphasizes the use of registers. Therefore, register use has significant impact on program performance. For example, fetching a value from a register is significantly faster than fetching a value from storage. Thus, the optimizer makes the best possible use of registers.

In allocating registers, the optimizer selects those data items most suited for registers, taking into account their frequency of use and their location in the program structure. In addition, the optimizer assigns values to registers so that their contents move

minimally within loops and during procedure invocations.

The primary benefits of optimization, of course, are faster running programs and smaller object code size. However, the optimizer can also speed up development time. For example, your coding time can be reduced if you let the optimizer relate programming details to execution time efficiency. This lets you focus on the more crucial global structure of your program. Moreover, programs often yield code sequences that can be optimized regardless of how well you write your source program.

6.2.2 General Considerations

When optimizing your program, consider the following:

- Optimize your programs only when they are fully developed and debugged. Although the optimizer does not alter the flow of control within a program, it may move operations so that the object code does not correspond to the source code. These changed sequences of code may create confusion when you use the debugger.
- The `-C` option of the Pascal compiler, which performs bounds checking in Pascal programs, inhibits some optimizations. Therefore, unless bounds checking is crucial, you should not specify the `-C` option when you optimize a Pascal program.
- Optimizations are most useful in program areas that contain loops. The optimizer moves loop-invariant code sequences outside loops so that they are performed only once instead of multiple times. Apart from loop-invariant code, loops often contain loop-induction expressions that can be replaced with simple increments. In programs composed of mostly loops, global optimization can often reduce the running time by half.

The following examples illustrate the results of loop optimization on source code that is compiled both with and without the `-O` compiler option. The source code is shown first, followed by the two outputs.

Source Code

```
void
left(a, distance)
  char a[];
  int distance;
  {
  int j, length;
  length = strlen(a) - distance;
  for (j = 0; j < length; j++)
    a[j] = a[j + distance];
  }
```

Unoptimized Code Output

```
#   8           for (j=0; j<length; j++)
        sw      $0, 36($sp)     # j = 0
        ble     $24, 0, $33     # length >= j
$32:
 #   9          a[j] = a[j+distance];
```

```
        lw      $25, 36($sp)    # j
        lw      $8, 44($sp)     # distance
        addu    $9, $25, $8     # j+distance
        lw      $10, 40($sp)    # address of a
        addu    $11, $10, $9    # address of a[j+
        lb      $12, 0($11)     # a[j+distance]
        addu    $13, $10, $25   # address of a[j]
        sb      $12, 0($13)     # a[j]
        lw      $14, 36($sp)    # j
        addu    $15, $14, 1     # j+1
        sw      $15, 36($sp)    # j++
        lw      $3, 32($sp)     # length
        blt     $15, $3, $32    # j < length
$33:
```

Optimized Code Output

```
 #   8          for (j=0; j<length; j++)
        move    $5, $0          # j = 0
        ble     $4, 0, $33      # length >= j
        move    $2, $16         # address of a[j]
        addu    $6, $16, $17    # address of a[j+distance]
$32:
 #   9          a[j] = a [j+distance];
        lb      $3, 0($6)       # a[j+distance]
        sb      $3, 0($2)       # a[j]
        addu    $5, $5, 1       # j++
        addu    $2, $2, 1       # address of next a[j]
        addu    $6, $6, 1       # address of next a[j+distance]
        blt     $5, $4, $32     # j < length
$33:
```

The optimized version contains fewer total instructions and fewer instructions that reference memory. Wherever possible, the optimizer replaces load and store instructions (which reference memory) with the faster computational instructions that perform operations only in registers.

6.2.3 Optimizing Separate Compilation Units

The optimizer processes one procedure at a time. Large procedures offer more opportunities for optimization, because more interrelationships are exposed in terms of constructs and regions. However, large procedures require more time to optimize than smaller ones.

The `uload` and `umerge` phases of the compilation permit global optimization among separate units in the same compilation. Often, programs are divided into separate files, called modules or compilation units, which are compiled separately. This saves compile time during program development because a change requires recompilation of only one compilation unit rather than the entire program.

Traditionally, program modularity restricted the optimization of code to a single compilation unit at a time rather than over the full breadth of the program. For example, calls to procedures that reside in other modules could not be fully optimized together with the code that called them.

The `uld` and `umerge` phases of the compiler system overcome this deficiency. The `uld` phase links multicompilation units into a single compilation unit. Then, `umerge` orders the procedures for optimal processing by the global optimizer (`uopt`).

6.2.4 Types of Optimization

The following sections describe these types of optimization:

- Full optimization
- Optimizing large programs
- Optimizing frequently used modules

For information about specific optimizing options, see `cc`(1), in the *ULTRIX Reference Pages and* `f77`(1) *and* `pc`(1) *in the Reference Pages for the FORTRAN and Pascal layered products.*

Figure 6-3 illustrates the major processing phases of the compiler and the way the `-O` compiler option determines the execution sequence.

6.2.4.1 Full Optimization – This section provides examples using the `-O3` compiler option. The examples provided in this section assume that the program `myprogram` consists of three files: `a.c`, `b.c`, and `c.c`.

To perform procedure merging optimizations (`-O3`) on all three files, you would type the following:

```
% cc -O3 -o foo a.c b.c c.c
```

If you normally use the `-c` option to compile the object file, follow these steps:

1. Compile each file separately using the `-j` option. For example:

```
% cc -j a.c
% cc -j b.c
% cc -j c.c
```

The `-j` option causes the compiler driver to produce a `.u` file (the standard compiler front-end output, which is made up of `ucode`, an internal language used by the compiler). None of the remaining compiling phases are executed, as is illustrated by the following:

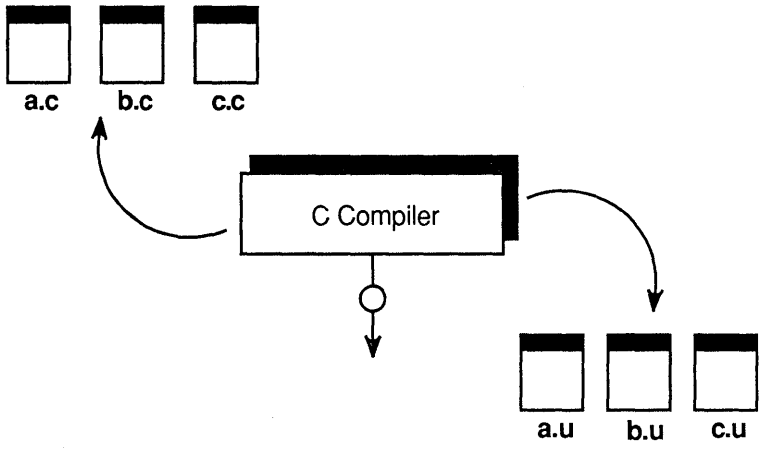
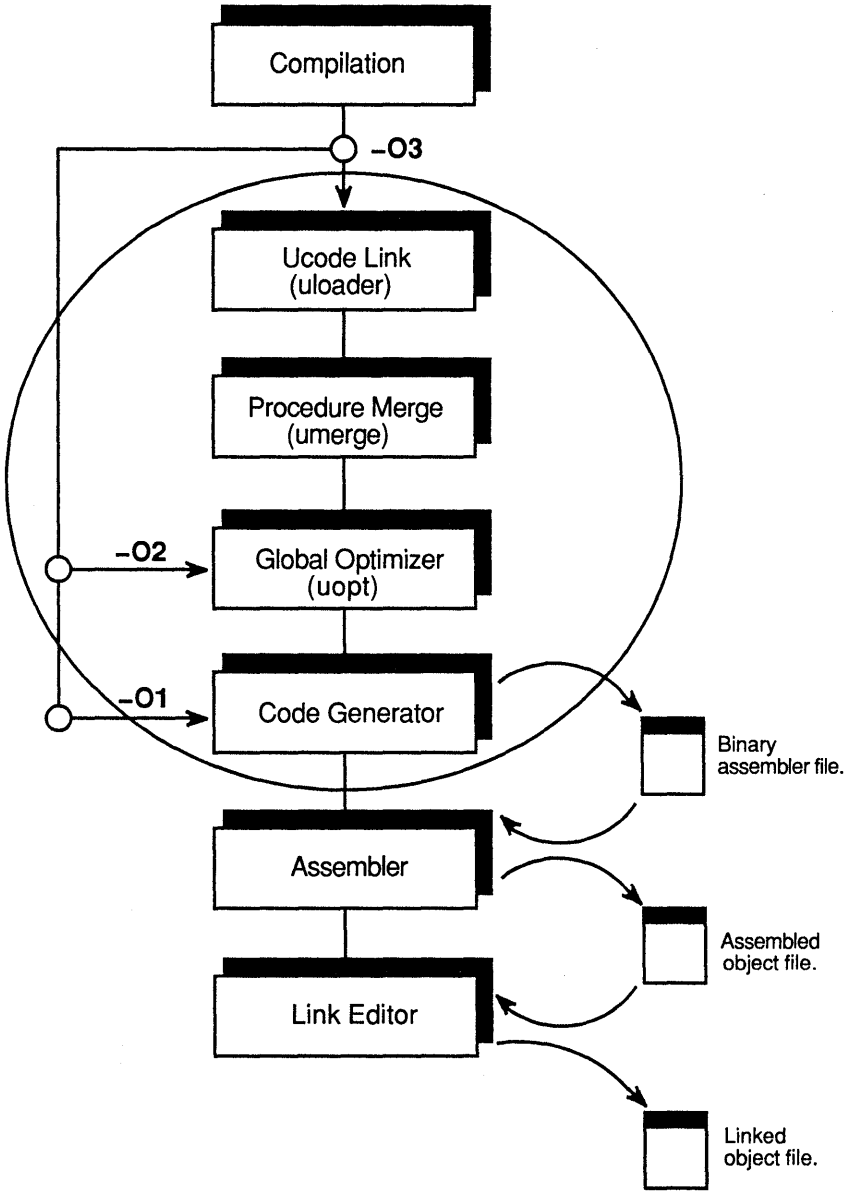


Figure 6-3 illustrates the optimization phases of the compiler.

Figure 6-3: Optimization Phases of the Compiler

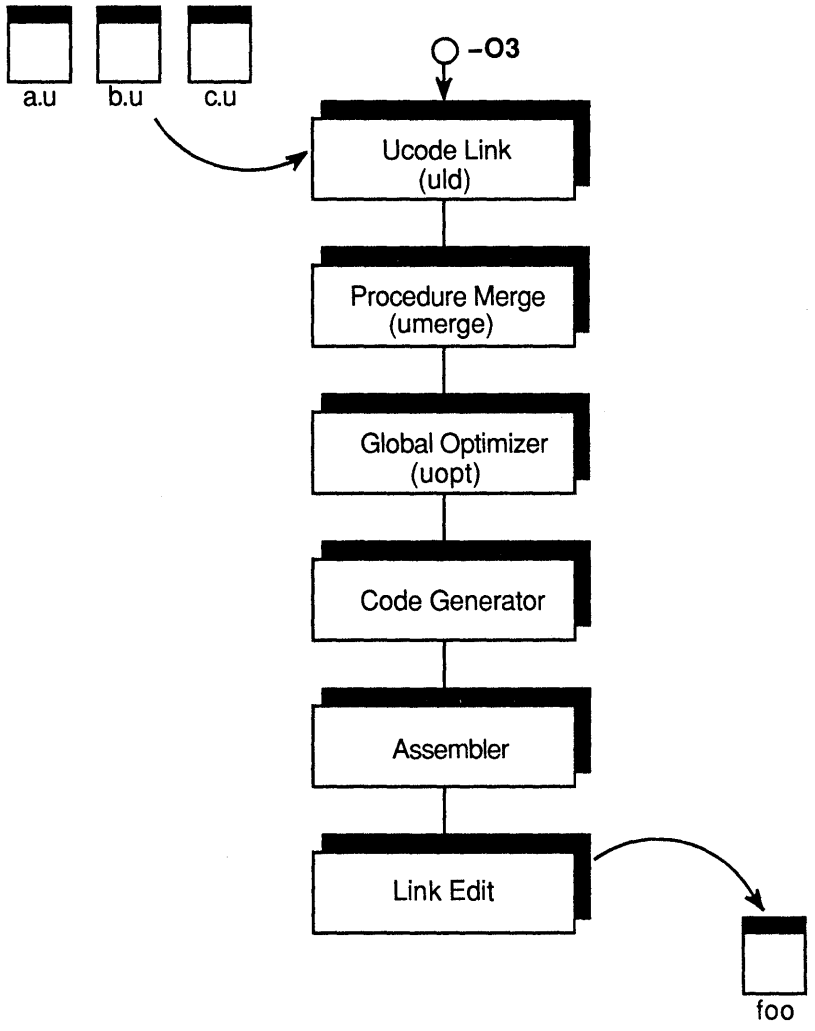


2. To perform optimization and complete the compilation process, enter the following:

```
% cc -O3 -o foo a.u b.u c.u
```

Figure 6-4 illustrates the results of executing this command.

Figure 6-4: –O3 Optimization



ZK-0072U-R

6.2.4.2 Optimizing Large Programs – To ensure that all program modules are optimized regardless of their size, specify the `-Olimit` option at compilation.

Because compilation time increases by the square of the program size, the compiler system enforces a top limit on the size of a program that can be optimized. This limit was set for the convenience of users who place a higher priority on the compilation turnaround time than on optimizing an entire program. The `-Olimit` option removes the top limit and lets those users who do not mind a long compilation to fully optimize their programs.

6.2.4.3 Optimizing Frequently Used Modules – You may want to compile and optimize modules that will be frequently called from programs written at a later time. This can reduce the compile and optimization time required when the modules are needed.

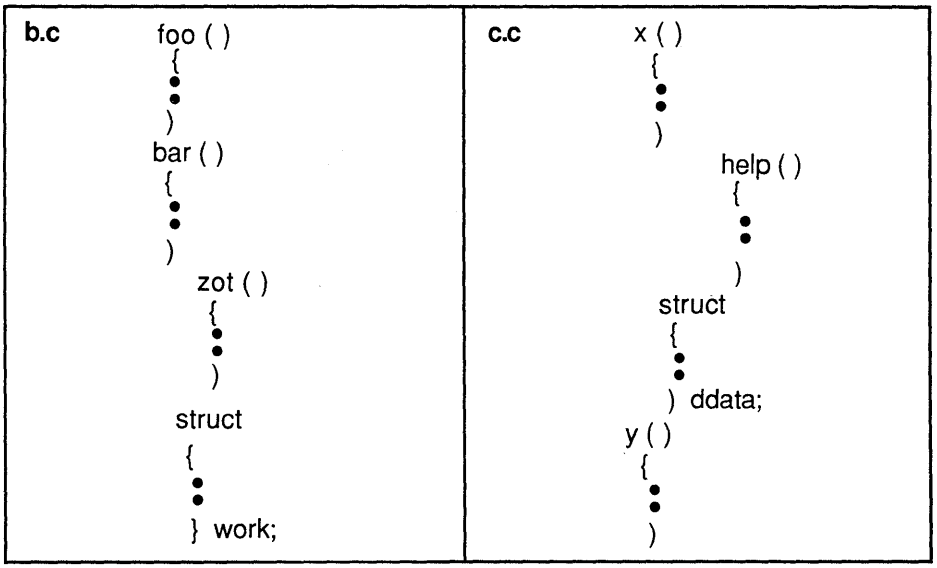
In the examples that follow, `b.c` and `c.c` represent two frequently used modules that are to be compiled and optimized, retaining all the necessary information to link them with later programs; `future.c` represents one such program.

1. Compile `b.c` and `c.c` separately. For example:

```
% cc -j b.c
% cc -j c.c
```

   The `-j` option causes the front end (first phase) of the compiler to produce two `ucode` files, `b.u` and `c.u`.

2. Create a file that contains the external symbols in `b.c` and `c.c` to which `future.c` will refer. Each symbolic name must be separated by at least one blank. The next figure provides the skeletal contents of `b.c` and `c.c`.



ZK-0074U-R

   The `future.c` program will call or reference only `foo`, `bar`, `x`, `ddata`, and `y` in the `b.c` and `c.c` procedures.

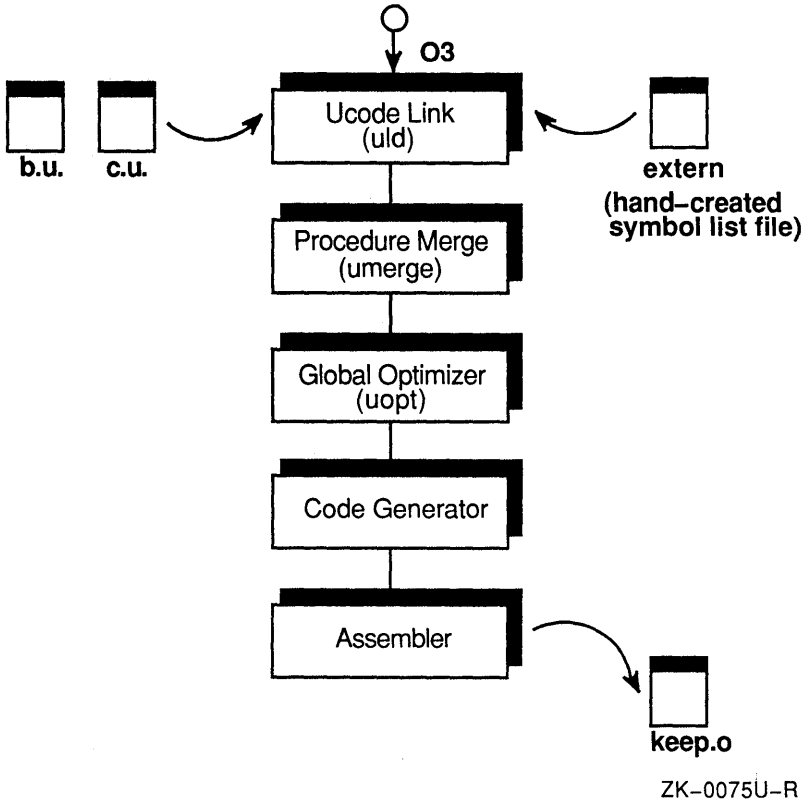
3. Create a file (named `extern` for this example) that contains the symbolic names `foo bar x ddata y`. (The structure (`work`) and the `help` and `zot` procedures are used internally only by `b.c` and `c.c`. Therefore, they are not included in `extern`.)

   If you omit an external symbolic name, an error message is generated (see step 5).

4. Optimize the `b.u` and `c.u` modules using the extern file. For example:

```
% cc -c -O3 -kp extern b.u c.u -o keep.o
```

   In the `-kp` option, the `k` designates that the `p` link editor option is to be passed to the `ucode` loader. The following figure illustrates this step.



ZK-0075U-R

5. Create a `ucode` file and an optimized object code file for `future.c`. For example:

```
% cc -O3 future.u keep.o -o foo
```

You may receive the following message, which indicates that the code in `future.c` is using a symbol from the code in `b.c` or `c.c` that was not specified in the `extern` file:

```
zot: multiply defined hidden external (should have been preserved)
```

If you receive this message, proceed to step 6.

6. Include `zot`, which the message indicates is missing, in the `extern` file and recompile. For example:

```
% cc -O3 -c -kp extern b.u c.u -o keep.o
% cc -O3 future.u keep.o -o foo
```

6.2.5 Building a ucode Object Library

Building a `ucode` object library is similar to building a Common Object File Format (`coff`) library. First, compile the source files into `ucode` object files using the `-j` compiler option and the archiver just as you would for `coff` object libraries. For example:

```
% cc -j a.c
% cc -j b.c
% cc -j c.c
% ar crs libfoo.b a.u b.u c.u
```

Conventional names exist for `ucode` object libraries (`libx.b`), just as they do for `coff` object libraries (`libx.a`).

6.2.6 Using ucode Object Libraries

Using `ucode` object libraries is similar to using `coff` object libraries. To load from a `ucode` library, specify a `-klx` compiler or `ucode` loader option. The following example loads the file created in the previous example from the ucode library:

```
% cc -O3 file1.u file2.u -klfoo -o output
```

Because the libraries are searched as they are encountered on the command line, the order in which you specify them is important. If a library is made from both assembly and high-level language routines, the `ucode` object library contains code only for the high-level language routines and not all the routines as the `coff` object library. In this case, you must specify to the `ucode` loader both the `ucode` object library and the `coff` object library, in that order, to ensure that all modules are loaded from the proper library.

If the compiler driver is to perform both a `ucode` load step and a final load step, the object file created after the `ucode` load step is placed in the position of the first `ucode` file specified or created on the command line in the final load step.

6.2.7 Improving FORTRAN Program Optimization

The following recommendation can help increase optimizing opportunities for the global optimizer (`uopt`):

- Avoid indirect calls (calls that use routines or pointers to functions as arguments).

  Indirect calls cause unknown side effects (that is, change global variables) that can reduce the amount of optimization.

The global optimizer processes programs only when you explicitly specify the `-O2` or `-O3` compiler option. However, the code generator and assembler phases of the compiler always perform certain optimizations (certain assembler optimizations are bypassed when you specify the `-O0` compiler option).

The following recommendations can help increase optimizing opportunities for the other passes of the compiler:

- As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers, where they remain during execution of the called routine. Therefore, always declare as the first four parameters those variables that are most frequently manipulated in the called routine with floating-point parameters preceding nonfloating-point parameters.
- Use word-size variables instead of smaller ones if enough space is available. This may take more space, but it is more efficient.

6.2.8 Improving C Program Optimization

The following recommendations can help increase optimizing opportunities for the global optimizer (`uopt`):

- Avoid indirect calls (calls that use routines or pointers to functions as arguments).

  Indirect calls cause unknown side effects (that is, change global variables) that can reduce the amount of optimization.

- Function return values

  Use function return values instead of reference parameters.

- Do, while, and repeat

  Use `do while` instead of `while` or `for` when possible. Then, the optimizer does not have to duplicate the loop condition to move code from within the loop to outside the loop.

- Unions and variant records

  Avoid unions that cause overlap between integer and floating point data types. This keeps the optimizer from assigning the fields to registers.

- Use local variables

  Avoid using global variables. Minimizing the use of global variables increases optimization opportunities for the compiler. Declare any variable outside of a function as static, unless that variable is referenced by another source file.

- Value parameters

  Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers.

- Pointers and aliasing

  Avoid using aliases by introducing local variables to store dereferenced results. (A dereferenced result is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, whereas local variables are not. Therefore, they can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing lets the compiler produce better code:

```
Source code:
int len = 10;
char a[10];
void
zero()
  {
  char *p;
  for (p =a; p != a +len; ) *p++ = 0;
  }
Generated assembly code:
# 8   for (p = a; p != a + len; ) *p++ = 0;
        move    $2, $4          # p = a
        lw      $3, len
        addu    $24, $4, $3
        beq     $24, $4, $33    # a + len != a
$32:
        sb      $0, 0($2)       # *p = 0
        addu    $2, $2, 1       # p++
        lw      $25, len
        addu    $8, $4, $25
        bne     $8, $2, $32     # len + a != p
$33:
```

To increase the efficiency of this example, you can use one of two methods:

- Use subscripts instead of pointers
- Use local variables to store unchanging values

- Use subscripts instead of pointers.

  The use of subscripting in the procedure `azero` eliminates aliasing; the compiler keeps the value of `len` in a register, which saves two instructions, and still uses a pointer to access *a* efficiently, even though a pointer is not specified in the source code. For example:

```
Source code:
void
azero()
  {
  int i;
  for (i = 0; i != len; i++) a[i] = 0;
  }
Generated assembly code:
#  14     for (i = 0; i != len; i++) a[i] = 0;
        move    $2, $0          # i = 0
        beq     $3, 0, $35      # len != 0
        la      $14, a
        move    $2, $14
        addu    $4, $3, $14     # a[len]
$34:
        sb      $0, 0($2)       # *a = 0
        addu    $2, $2, 1       # a++
        bne     $2, $4, $34     # a != a[len]
$35:
```

- Use local variables.

  Specifying `len` as a local variable or formal argument ensures that aliasing can not take place and permits the compiler to place len in a register. For example:

```
Source code:
char a[10];
void
lpzero(len)
  int len;
  {
  char *p;
  for (p = a; p != a + len; ) *p++ = 0;
  }
Generated assembly code:
#   8      for (p = a; p != a + len; ) *p++ = 0;
        move    $2, $6              # p = a
        addu    $5, $6, $4
        beq     $5, $6, $33         # a + len != a
$32:
        sb      $0, 0($2)           # *p = 0
        addu    $2, $2, 1           # p++
        bne     $5, $2, $32         # a + len != p
$33:
```

  In the previous example, the compiler generates slightly more efficient code for the second method.

- Write straightforward code.

For example, do not use autoincrement (++) and autodecrement (--) operators within an expression. When you use these operators for their values, rather than for their side effects, you often get bad code. For example:

```
Bad:
while (n--) {
    .
    .
    .
}
Good:
while (n != 0) {
  n--;
    .
    .
    .
}
```

- Use register declarations liberally.

  The compiler automatically assigns variables to registers. However, specifically declaring a register type lets the compiler make more aggressive assumptions when assigning register variables.

- Avoid taking and passing addresses. This can create aliases, make the optimizer store variables from registers to their home storage locations, and significantly reduce optimization opportunities that would otherwise be performed by the compiler.

- VARARGs

  Avoid functions that take a variable number of arguments. This causes the optimizer to unnecessarily save all parameter registers on entry.

The global optimizer processes programs only when you explicitly specify the `-O2` or `-O3` compiler option. However, the code generator and assembler phases of the compiler always perform certain optimizations (certain assembler optimizations are bypassed when you specify the `-O0` compiler option).

The following recommendations can help increase optimizing opportunities for the other passes of the compiler:

- Use tables rather than if-then-else or switch statements. For example:

```
Good:
if ( i == 1 ) c = '1';
 else c = '0';
More efficient:
c = "01"[i];
```

- As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers, where they remain during execution of the called routine. Therefore, always declare as the first four parameters those variables that are most frequently manipulated in the called routine with floating-point parameters preceding nonfloating-point parameters.

- Use word-size variables instead of smaller ones if enough space is available. This may take more space, but it is more efficient.

- Rely on `libc` functions (for example, `strcpy`, `strlen`, `strcmp`, `bcopy`, `bzero`, `memset`, and `memcpy`). These functions were hand-coded for efficiency.

- Use the unsigned data type for variables wherever possible for the following reasons:
  - Because it knows the variable will always be greater than or equal to zero (>=0), the compiler can perform optimizations that would not otherwise be possible.
  - The compiler generates fewer instructions for multiply and divide operations that use a power of 2.

    For example:

```
int i;
unsigned j;
     .
     .
     .
return i/2 + j/2;
```

    The compiler generates four instructions for the signed i/2 operations:

```
000000 bgez     r14, 0xC
000004 move     r1, r14
000008 addiu    r1, r1, 1
00000c sra      r15, r1, 1
```

    By contrast, the compiler generates only one instruction for the unsigned j/2 operation:

```
000010 srl      r24,r5,1    # j / 2
```

    In this example, i/2 is an expensive expression, and j/2 is an inexpensive one.

6.2.9 Improving Pascal Program Optimization

The following recommendations can help increase optimizing opportunities for the global optimizer (`uopt`):

- Avoid indirect calls (calls that use routines or pointers to functions as arguments).

  Indirect calls cause unknown side effects (that is, change global variables) that can reduce the amount of optimization.

- Function return values

  Use function return values instead of reference parameters.

- Do, while, and repeat

  Use `repeat` instead of `while` or `for` when possible. Then, the optimizer does not have to duplicate the loop condition to move code from within the loop to outside the loop.

- Variant records

  Avoid variant records that cause overlap between integer and floating point data types. This keeps the optimizer from assigning the fields to registers.

- Use local variables

Avoid using global variables. Minimizing the use of global variables increases optimization opportunities for the compiler.

- Value parameters

  Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers.

- Use packed arrays only when space is crucial. Packed arrays prevent moving induction expressions from within a loop to outside the loop.

The global optimizer processes programs only when you explicitly specify the `-O2` or `-O3` compiler option. However, the code generator and assembler phases of the compiler always perform certain optimizations (certain assembler optimizations are bypassed when you specify the `-O0` compiler option).

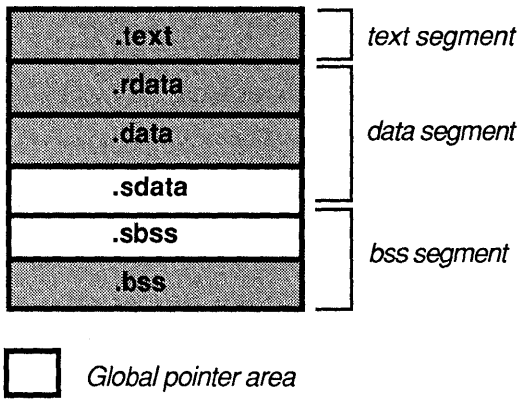
The following recommendations can help increase optimizing opportunities for the other passes of the compiler:

- As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers, where they remain during execution of the called routine. Therefore, always declare as the first four parameters those variables that are most frequently manipulated in the called routine with floating-point parameters preceding nonfloating-point parameters.
- Use word-size variables instead of smaller ones if enough space is available. This may take more space, but it is more efficient.
- Use predefined functions as much as possible. For example:
  - max and min rather than if-then-else
  - Shift and bitwise `and` instead of `div` and `mod`

6.3 Controlling the Size of Global Pointer Data

This section describes the global pointer area and how, by controlling the size of variables and constants that the compiler places in this area, you can improve program performance.

Global pointer data are constants and variables that the compiler places in the `.sdata` and `.sbss` portions of the `data` and `bss` segments shown in the following figure. This area is referred to as the global pointer area.



ZK-0076U-R

(The `.rdata`, `.data`, and `.sdata` sections contain initialized data, and the `.sbss` and `.bss` sections reserve space for uninitialized data that is created by the kernel loader for the program before execution and filled with zeros.)

In general, the compiler system emits two machine instructions to access a global value. However, by using a register as a global pointer (called `$gp`), the compiler creates the 65,536-byte global pointer area where a program can access any value with a single machine instruction – only half the number of instructions required without a global pointer.

To maximize the number of individual variables and constants that a program can access in the global pointer area, the compiler first places those variables and constants that take the fewest bytes of memory. By default, the variables and constants occupying eight or fewer bytes are placed in the global pointer area, and those occupying more than eight bytes are placed in the `.data` and `.bss` sections.

6.3.1 Limiting the Size of Global Pointer Data

The more data that the compiler places in the global pointer area, the faster a program executes. However, if the data to be placed in the global pointer area exceeds 65,536 bytes, the link editor prints an error message and does not create an executable object file. In this case, you need to use the `-G` option to reduce the use of global data.

For most programs, the 8-byte default produces optimal results. However, the compiler provides the `-G` option to let you change the default size. For example:

```
-G 12
```

This causes the compiler to place only those variables and constants that occupy 12 or fewer bytes in the global pointer area.

6.3.2 Obtaining Optimal Global Data Size

The compiler places some variables in the global pointer area regardless of the setting of the `-G` option. For example, a program written in assembly language may contain `.sdata` directives that cause variables and constants to be placed into the global pointer area regardless of size. Moreover, the `-G` option does not affect variables and constants in libraries and objects compiled beforehand.

To alter the allocation size for the global pointer area for data from these objects, you must recompile them and specify the -G option and the desired value.

Thus, two potential problems exist in specifying a maximum size in the -G option:

- Using a value that is too small can reduce the speed of the program.
- Using a value that is too large can cause more than the maximum of 65,536 bytes to be placed in the data area, which creates an error condition and produces an unexecutable object module.

The -bestGnum link editor option helps overcome these problems by predicting an optimal value to specify for the -G option. The following sections provide examples of using the -bestGnum option and the related -nocount and -count options.

6.3.2.1 Examples (Excluding Libraries) – When you use the -bestGnum option exclusive of -nocount and -count, the compiler assumes that you cannot recompile any libraries to which it would link automatically and causes the link editor not to consider these libraries when predicting the optimal maximum size. However, if you link to other system-supplied libraries, you must specify -nocount before the library. For example:

```
% cc -bestGnum foo.c -nocount -lm
% pc -bestGnum foo.p
```

In the second command, the compiler produces a message that provides the best value for -G. If all program data fits into the global pointer area, a message similar to the following indicates this fact:

```
All data will fit into the global pointer area
Best -G num value to compile with is 80 (or greater)
```

Because all data fits into the global pointer area, no recompilation is necessary. Consider the following example, which specifies 70000 as the maximum size of a data item to be placed in the global pointer area:

```
% pc ersatz.p -G 70000 -bestGnum
```

This example produces the following messages:

```
gp relocation out-of-range errors have occurred and bad object file
produced (corrective action  must be taken)
Best -G num value to compile with is 1024
```

In this example, the link editor does not produce an executable load module and recommends a recompilation as follows:

```
% pc real.p -G 1024
```

6.3.2.2 Example (Including Libraries) – You can explicitly specify that the link editor either include or exclude specific libraries in predicting the -G value. For example:

```
% cc -o plotter -bestGnum plotter.o -nocount libieee.a \
-count liblaser.a
```

In this example, the link editor assumes that libieee.a cannot be recompiled and will continue to occupy the same space in the global pointer area. It assumes that plotter.o and liblaser.a can be recompiled and produces a recommended -G value to use on recompilation.

Improving Program Performance (VAX) 7

This chapter describes facilities that can help reduce the execution time of your programs on VAX processors. It discusses the following topics:

- Profiling code
- Optimizing code

The best way to produce efficient code is to follow good programming practices:

- Choose good algorithms and leave the details to the compiler.
- Avoid tailoring your work for any particular release or quirk of the compiler system.

7.1 Profiling Code

The profiler isolates those portions of your code where execution is concentrated and provides reports that indicate where you should devote your time and effort for coding improvements. This section describes the advantages of the profiler and how to use it.

In a typical program, execution time is confined to a relatively few sections of code, and it is profitable to concentrate on improving coding efficiency in only those sections. The compiler system provides the following profile information:

- Program counter (pc) sampling
- Invocation counting

The program counter highlights the execution time spent in various parts of the program. You can obtain pc sampling information by compiling the desired source modules with the `-p` option and then executing the resulting program object, which generates profile data in raw format. Your program must exit normally for the profile data to be created.

Invocation counting gives the number of times each procedure in the program is invoked. You can obtain invocation counting information by compiling the desired source modules using the `-pg` option, which uses your source program to create an equivalent program that contains additional code that counts the execution of each function. Executing the equivalent program generates the profile data in raw format.

In addition, by using the `prof` and `gprof` commands, you can create a formatted listing of the raw profile data. You can use this listing to determine if your program exercised all portions of your code, to determine where to correct inefficient code, or to determine where to substitute better algorithms or assembly code.

The following is an example of a pc sampling listing that was produced from a program compiled with the `-p` compiler option. The program was then executed. The `prof` command produced the listing from the raw profile data generated during

execution. The output is sorted in descending order by the total time spent in each procedure; unexecuted procedures are excluded. For further information, see `prof`(1) in the *ULTRIX Reference Pages.*

```
Each sample covers 8.00 byte(s) for 4.2% of 0.2400 seconds
 %time  cumsecs  #call  ms/call  name
  40.6     0.22     73     2.97  _write
  15.6     0.30      1    83.34  _yyparse
  12.5     0.37                  mcount
   6.3     0.40    105     0.32  __doprnt
   6.3     0.43                  _access
   6.3     0.47    617     0.05  _yylook
   3.1     0.48   2376     0.01  __flsbuf
   3.1     0.50                  _fprintf
   3.1     0.52    771     0.02  _malloc
   3.1     0.53      2     8.33  _read
   0.0     0.53      2     0.00  __filbuf
   0.0     0.53      2     0.00  __getstdiobuf
   0.0     0.53    173     0.00  _cat
   0.0     0.53      9     0.00  _docast
   0.0     0.53      6     0.00  _docexplain
   0.0     0.53      9     0.00  _dodeclare
   0.0     0.53      9     0.00  _dodexplain
   0.0     0.53      1     0.00  _dohelp
   0.0     0.53     39     0.00  _doprompt
   0.0     0.53      1     0.00  _doset
   0.0     0.53      1     0.00  _dostdin
   0.0     0.53    596     0.00  _ds
   0.0     0.53      1     0.00  _fflush
   0.0     0.53    696     0.00  _free
   0.0     0.53      2     0.00  _fstat
   0.0     0.53      1     0.00  _getopt
   0.0     0.53      2     0.00  _ioctl
   0.0     0.53      2     0.00  _isatty
   0.0     0.53      1     0.00  _main
   0.0     0.53     60     0.00  _mbcheck
   0.0     0.53    105     0.00  _printf
   0.0     0.53      1     0.00  _profil
   0.0     0.53     39     0.00  _prompt
   0.0     0.53      1     0.00  _rindex
   0.0     0.53     14     0.00  _sbrk
   0.0     0.53      1     0.00  _setprogname
   0.0     0.53    615     0.00  _strcat
   0.0     0.53     62     0.00  _strcmp
   0.0     0.53    596     0.00  _strcpy
   0.0     0.53   1314     0.00  _strlen
   0.0     0.53     64     0.00  _strncmp
   0.0     0.53    404     0.00  _yylex
   0.0     0.53      1     0.00  _yywrap
```

The next example shows `gprof` command listings.

```
granularity: each sample hit covers 4 byte(s) for 2.17% of 0.46 seconds
%time cumsecs seconds   calls name
 39.1    0.18    0.18      73 _write
 15.2    0.25    0.07         mcount
 13.0    0.31    0.06     617 _yylook
 10.9    0.36    0.05       1 _yyparse
  6.5    0.39    0.03     596 _ds
  4.3    0.41    0.02    2376 __flsbuf
  4.3    0.43    0.02     105 __doprnt
  2.2    0.44    0.01     771 _malloc
  2.2    0.45    0.01     615 _strcat
  2.2    0.46    0.01     596 _strcpy
  0.0    0.46    0.00    1314 _strlen
```

```
  0.0     0.46     0.00      696 _free
  0.0     0.46     0.00      404 _yylex
  0.0     0.46     0.00      173 _cat
  0.0     0.46     0.00      105 _printf
  0.0     0.46     0.00       64 _strncmp
  0.0     0.46     0.00       62 _strcmp
  0.0     0.46     0.00       60 _mbcheck
  0.0     0.46     0.00       39 _doprompt
  0.0     0.46     0.00       39 _prompt
  0.0     0.46     0.00       15 _sbrk
  0.0     0.46     0.00        9 _docast
  0.0     0.46     0.00        9 _dodeclare
  0.0     0.46     0.00        9 _dodexplain
  0.0     0.46     0.00        7 _morecore
  0.0     0.46     0.00        6 _docexplain
  0.0     0.46     0.00        2 __filbuf
  0.0     0.46     0.00        2 __getstdiobuf
  0.0     0.46     0.00        2 _fstat
  0.0     0.46     0.00        2 _ioctl
  0.0     0.46     0.00        2 _isatty
  0.0     0.46     0.00        2 _read
  0.0     0.46     0.00        1 _dohelp
  0.0     0.46     0.00        1 _doset
  0.0     0.46     0.00        1 _dostdin
  0.0     0.46     0.00        1 _fflush
  0.0     0.46     0.00        1 _getopt
  0.0     0.46     0.00        1 _main
  0.0     0.46     0.00        1 _profil
  0.0     0.46     0.00        1 _rindex
  0.0     0.46     0.00        1 _setprogname
  0.0     0.46     0.00        1 _yywrap
granularity: each sample hit covers 4 byte(s) for 2.56% of 0.39 seconds
                                called/total       parents
index  %time    self descendents  called+self    name        index
                                called/total       children
                                                   <spontaneous>
[1]    100.0    0.00        0.39                 start [1]
                0.00        0.39       1/1           _main [3]
-----------------------------------------------
                0.00        0.39       1/1           _main [3]
[2]    100.0    0.00        0.39       1         _dostdin [2]
                0.05        0.34       1/1           _yyparse [4]
                0.00        0.00       1/2           _isatty [39]
-----------------------------------------------
                0.00        0.39       1/1           start [1]
[3]    100.0    0.00        0.39       1         _main [3]
                0.00        0.39       1/1           _dostdin [2]
                0.00        0.00       1/1           _setprogname [45]
                0.00        0.00       1/62          _strcmp [31]
                0.00        0.00       1/1           _getopt [42]
-----------------------------------------------
                0.05        0.34       1/1           _dostdin [2]
[4]    100.0    0.05        0.34       1         _yyparse [4]
                0.00        0.07     404/404         _yylex [9]
                0.00        0.06       9/9           _dodexplain [11]
                0.00        0.05       1/1           _dohelp [12]
                0.02        0.01     480/596         _ds [13]
                0.00        0.04       9/9           _dodeclare [14]
                0.00        0.03       1/1           _doset [15]
                0.00        0.03       6/6           _docexplain [16]
                0.00        0.02       9/9           _docast [17]
                0.00        0.01     173/173         _cat [18]
                0.00        0.00      79/1314        _strlen [28]
                0.00        0.00      64/64          _strncmp [30]
                0.00        0.00      60/60          _mbcheck [32]
```

```
                0.00        0.00       39/39          _prompt [34]
                0.00        0.00       39/39          _doprompt [33]
                0.00        0.00       32/62          _strcmp [31]
-----------------------------------------------
                0.02        0.20      105/105         _printf [6]
[5]     56.4    0.02        0.20      105           __doprnt [5]
                0.02        0.18     2376/2376        __flsbuf [7]
-----------------------------------------------
    ... many additional lines deleted ...
```

7.1.1 PC Sampling

To obtain pc sampling information, follow this procedure:

1. Compile and link-edit your program with the `-p` option. For example:

   ```
   % cc -c -p myprog.c
   % cc -p -o myprog myprog.o
   ```

   Note that you must specify the `-p` profiling option during both the compilation and link-editing steps to obtain pc sampling information.

2. Execute the profiled program. During its execution, profiling data is saved in the profile data file. The default is `mon.out`.

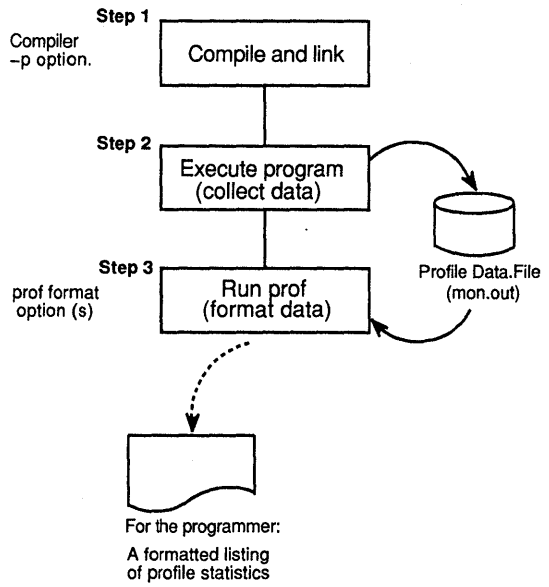
3. Run the `prof` profile formatter, which extracts information from each profile data file and formats it in an easily readable form. For example:

   ```
   % prof -procedure myprog mon.out
   ```

   For further information, see `prof`(1) in the *ULTRIX Reference Pages*.

Figure 7-1 illustrates the procedure for obtaining pc sampling information.

Figure 7-1: PC Sampling



ZK-0176U-R

7.1.2 Running the prof Profiler

The `prof` profiler converts the raw profiling information into either a printed listing or an output file that can be used by the compiler. To execute the profiler, use the following syntax:

prof [*options*] [*file*]

If you do not specify a profile data file, `prof` looks for `mon.out` in the current directory. For further information, see `prof`(1) in the *ULTRIX Reference Pages*.

7.2 Optimizing Code

The following sections provide an overview of the major optimizing techniques. They also give examples showing optimization techniques.

7.2.1 General Considerations

When optimizing your program, consider the following:

- Optimize your programs only when they are fully developed and debugged. Although the optimizer does not alter the flow of control within a program, it may move operations so that the object code does not correspond to the source code. These changed sequences of code may create confusion when you use the debugger.
- The `-C` option of the Pascal compiler, which performs bounds checking in Pascal programs, inhibits some optimizations. Therefore, unless bounds checking is crucial, you should not specify the `-C` option when you optimize a Pascal program.

7.2.2 Improving C Program Optimization

The following recommendations can help increase optimizing opportunities for the optimizer (`c2`).

- Avoid indirect calls (calls that use routines or pointers to functions as arguments).

  Indirect calls cause unknown side effects (that is, change global variables) that can reduce the amount of optimization.

- Function return values

  Use function return values instead of reference parameters.

- Unions

  Avoid unions that cause overlap between integer and floating point data types. This keeps the optimizer from assigning the fields to registers.

- Use local variables

  Avoid using global variables. Minimizing the use of global variables increases optimization opportunities for the compiler. Declare any variable outside of a function as static, unless that variable is referenced by another source file.

- Value parameters

Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers.

- Pointers and aliasing

  Avoid using aliases by introducing local variables to store dereferenced results. (A dereferenced result is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, whereas local variables are not. Therefore, they can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing lets the compiler produce better code:

```
Source code:
int len = 10;
char a[10];
void
zero()
  {
  register char *p;
  for (p =a; p != a +len; ) *p++ = 0;
  }
Generated assembly code:
# for (p = a; p != a+len; ) *p++ = 0;
        moval   _a,r11          # p = a
        jbr     L20
L2000001:
        clrb    (r11)+          # *p++ = 0
L20:    addl3   _len,$_a,r0     # a+len
        cmpl    r11,r0          # p != a+len
        jneq    L2000001
```

  To increase the efficiency of this example, you can use one of two methods:

  - Use subscripts instead of pointers
  - Use local variables to store unchanging values

- Use subscripts instead of pointers.

  The use of subscripting in the procedure `azero` eliminates aliasing; the compiler keeps the value of `len` in a register, which saves two instructions, and still uses a pointer to access *a* efficiently, even though a pointer is not specified in the source code. For example:

```
Source code:
void
azero()
  {
  register int i;
  for (i = 0; i != len; i++) a[i] = 0;
  }
Generated assembly code:
# for (i = 0; i != len; i++) a[i] = 0;
        clrl    r11             # i = 0
        jbr     L20
L2000001:
        clrb    _a[r11]         # a[i] = 0
        incl    r11             # i++
L20:    cmpl    r11,_len        # i != len
        jneq    L200000
```

- Use local variables.

Specifying `len` as a local variable or formal argument ensures that aliasing can not take place and permits the compiler to place len in a register. For example:

```
Source code:
char a[10];
void
lpzero(len)
  register int len;
  {
  register char *p;
  for (p = a; p != a + len; ) *p++ = 0;
  }
Generated assembly code:
# for (p = a; p != a+len; ) *p++ = 0;
        movl    4(ap),r11       # p = a
        moval   _a,r10          # register p
        jbr     L19
L2000001:
        clrb    (r10)+          # *p++ = 0
L19:    addl3   r11,$_a,r0      # a+len
        cmpl    r10,r0          # p != a+len
        jneq    L2000001
```

In the previous example, the compiler generates slightly more efficient code for the second method.

- Write straightforward code.

  For example, do not use autoincrement (++) and autodecrement (--) operators within an expression. When you use these operators for their values, rather than for their side effects, you often get bad code. For example:

```
Bad:
while (n--) {
    .
    .
    .
}
Good:
while (n != 0) {
  n--;
    .
    .
    .
}
```

- Use register declarations liberally.

  The compiler will not place a variable in a register unless directed to do so.

The optimizer processes programs only when you explicitly specify the `-O` option. However, the code generator phase of the compiler always performs certain optimizations. The following recommendations can help increase these optimizing opportunities.

- Use tables rather than if-then-else or switch statements. For example:

```
Good:
if ( i == 1 ) c = '1';
 else c = '0';
More efficient:
c = "01"[i];
```

- Rely on `libc` functions (for example, `strcpy`, `strlen`, `strcmp`, `bcopy`, `bzero`, `memset`, and `memcpy`). These functions were hand-coded for efficiency.

Debugging 8

The ULTRIX programming environment provides many debugging tools. This chapter shows typical uses for these tools, taking you through a session with each as the tool is used on a sample program. The following tools are shown:

- `ctrace`: allows you to watch program flow and observe changes to variables
- `dbx`: invokes an interactive debugger
- `error`: inserts error messages from a compiler or language processor into a source file at the point of error
- `gcore`: creates a core image file of a running process
- `lint`: checks C source files for waste, errors, and nonportable code
- `trace`: traces the system calls made by a command

Each tool is discussed in its own section, organized as follows:

- Description
- Example
- Details

In addition, two other sections deal with kernel debugging; one section for RISC systems, one for VAX systems.

ULTRIX also provides an interactive debugger with a window interface, `dxdb`. For information on the `dxdb` debugger, see the *Guide to the dxdb Debugger*, in the ULTRIX Worksystem Software documentation.

The sample program used in this chapter is a simple editor that reads a line from stdin, performs some changes, and writes the modified line to stdout. This program is unimportant; you need not understand it to follow the examples. It is shown here for completeness:

```
/* This program is a crude editor that can make very simple changes
/* to lines of text.
*/

#include <stdio.h>
#define MAX 80

char *stredit (source, edit)
     char source[];
     int edit;
{
    char i, j;

    if ((edit - 16) >= 0)
    {  edit -= 16;
       for (i=0; i<=MAX; i++)
       {
           if ((source[i]==' ') && (source[i+1]==' '))
```

```
            {
                for(j=i; j<=MAX; j++)
                source[j]=source[j+1];
                --i;
            }
        }
    }

    if ((edit - 8) >= 0)
    {
        edit -= 8;
        for(i=0; i<=MAX; i++)
        {
            if (source[i]==' ')
            {
                for(j=i; j<=MAX; j++)
                source[j]=source[j+1];
                --i;
            }
        }
    }

    if ((edit - 4) >= 0)
    {
        edit -= 4;
        if ((source[0] >= 'a')&&(source[0] <= 'z'))
            source[0] -= ('a' - 'A');
        for(i=0; i<=MAX; i++)
            if ((source[i]==' ')&&(source[i+1] >= 'a')&&(source[i+1] <= 'z'))
        source[i+1] -= ('a' - 'A');
    }

    if ((edit - 2) >= 0)
    {
        edit -= 2;
        for(i=0; i<=MAX; i++)
            if ((source[i] >= 'A')&&(source[i] <= 'Z'))
                source[i] += ('a' - 'A');
    }

    if ((edit - 1) >= 0)
    {
        edit -= 1;
        for(i=0; i<=MAX; i++)
            if ((source[i] >= 'a')&&(source[i] <= 'z'))
                source[i] -= ('a' - 'A');
    }

    return(source);
}

getline (st)
char *st;
{
    char c;
    int i;

    for(i=0; i<=MAX ; i++)
    {
        st[i]=getchar();
        if (st[i]=='\n')
            break;
    }
    st[++i]='\0';
}
```

```
main()
{
    char str[MAX];
    int choice;

    printf("%s", "Enter a text line: ");
    getline(str);  printf("\n\n");

    printf("%s\n", "Choose an editing change or combination of changes,");
    printf("%s\n", "by entering a number or a sum of numbers.");
    printf("%s\n", "In the case of conflicting changes--for example, ");
    printf("%s\n", "\"3\" (UPPERCASE and lowercase)--the change with ");
    printf("%s\n", "the lower number will prevail.");
    printf("\n");
    printf("%s",   "    1  UPPERCASE");
    printf("%s\n", "           (highest priority)");
    printf("%s\n", "    2  lowercase");
    printf("%s\n", "    4  Initial Capital On All Words");
    printf("%s\n", "    8  No_blanks");
    printf("%s",   "    16 No excess blanks");
    printf("%s\n", "       (lowest priority)");
    printf("\n");
    printf("%s", "Enter your choice: ");
    scanf("%d", &choice);  printf("\n");

    printf("\n%s\n", stredit(str, choice));
}
```

8.1 ctrace

ctrace [*options*] [*input_file* [> *output_file*]]

8.1.1 Description

The `ctrace` command allows you to watch program flow and observe changes to variables, looking for unexpected behavior. Running `ctrace` on a source file places additional code into the file; this code causes executable statements and referenced or modified variables and their values to be written to stdout during the program's execution. Your source file must compile without errors before you use `ctrace` on it.

8.1.2 Example

```
% ctrace crude_editor.c > temp.c    # Direct the expanded code to a file.
% cc temp.c                         # Compile the expanded code.
% a.out                             # Run it.
 85 main()
 90     printf("%s", "Enter a text line: ");Enter a text line:
 91     getline(str);
        /* str == 2147475548 */
 70 getline (st)
 76     for(i=0; i<=MAX ; i++)
        /* i == 0 */
        /* MAX == 80 or 'P' */
 77     {
 78         st[i]=getchar();
            /* i == 0 */
```

The numbers to the left of the preceding source lines are the line numbers relative to the file `crude_editor.c`. Code lines displayed without numbers are lines added by `ctrace`; notice that these lines look like comments but actually display information about the contents of the variable referenced in the preceding line. The output pauses after line 78 while getchar( ) waits for input.

Loops are detected by `ctrace`, which displays the looping code only once but tells how many repetitions occur, as shown in the following portion of the display taken from later in the program's execution:

```
 62         for(i=0; i<=MAX; i++)
            /* i == 23 */
            /* MAX == 80 or 'P' */
 63             if ((source[i] >= 'a')&&(source[i] <= 'z'))
                /* i == 23 */
                /* source[i] == 0 */
    /* repeating */
    /* repeated 57 times */
```

8.1.3 Details

Complete information on `ctrace`'s options can be found in the *ULTRIX Reference Pages*

8.1.3.1 Tracing Only Certain Functions – Sifting through the trace of a large program is tedious. Moreover, many times you can isolate a problem to certain functions or certain sections of code. To discriminate among functions, use the **-f** and **-v** options on the `ctrace` command line:

-f *functions*	Trace only these functions
-v *functions*	Trace all functions except these

For example:

```
% ctrace -f getline main crude_editor.c > traced.c
```

The preceding command line creates the file `traced.c`, which (when compiled and run) shows a trace of the functions getline( ) and main( ). The following command line produces the file `traced.c`, which (when compiled and run) shows a trace of the entire program *except* the functions getline( ) and main( ):

```
% ctrace -v getline main crude_editor.c > traced.c
```

8.1.3.2 Tracing Only Certain Sections of Code – To trace only certain sections of code, insert the ctroff( ) and ctron( ) functions around code you do not want to trace. The ctroff( ) and ctron( ) functions turn `ctrace` off and on, respectively; for example:

```
main()
{
    char str[MAX];
    int choice;

    printf("%s", "Enter a text line: ");
    getline(str);  printf("\n\n");

ctroff();  /*****  Turn off tracing  *****/

    printf("%s\n", "Choose an editing change or combination of changes,");
    printf("%s\n", "by entering a number or a sum of numbers.");
    printf("%s\n", "In the case of conflicting changes--for example, ");
    printf("%s\n", "\"3\" \(UPPERCASE and lowercase\)--the change with ");
    printf("%s\n", "the lower number will prevail.");
    printf("\n");
    printf("%s",   "    1  UPPERCASE");
    printf("%s\n", "          \(highest priority\)");
    printf("%s\n", "    2  lowercase");
    printf("%s\n", "    4  Initial Capital On All Words");
    printf("%s\n", "    8  No_blanks");
    printf("%s",   "    16 No excess blanks");
    printf("%s\n", "       \(lowest priority\)");
    printf("\n");
    printf("%s", "Enter your choice: ");
    scanf("%d", &choice);  printf("\n");

ctron();  /*****  Turn tracing back on  *****/

    printf("\n%s\n", stredit(str, choice));
}
```

When run through `ctrace`, compiled, and executed, the preceding code fragment produces:

```
 93 ctroff();
    /* trace off */
Choose an editing change or combination of changes,
by entering a number or a sum of numbers.
In the case of conflicting changes--for example,
"3" (UPPERCASE and lowercase)--the change with
the lower number will prevail.

    1  UPPERCASE            (highest priority)
    2  lowercase
    4  Initial Capital On All Words
    8  No_blanks
    16 No excess blanks        (lowest priority)

Enter your choice: 20


    /* trace on */
114     printf("\n%s\n", stredit(str, choice));
```

Notice that all the code between the ctroff( ) and ctron( ) function calls still executes (producing the output in the example), but the code itself does not appear.

8.2 dbx

dbx [*options*] [*object_file* [*core_dump*]]

8.2.1 Description

The dbx command invokes the dbx debugger, which can:

- Display source code with line numbers
- Execute code conditionally
- Execute code one line at a time
- Execute code one machine instruction at a time
- Set and remove breakpoints
- Trace a line, a routine, or an entire program
- Trap signals sent to your program
- Call routines outside of normal program flow
- Examine a variable's content
- Assign a value to a variable
- Create command aliases
- Debug the ULTRIX kernel, /vmunix.

8.2.2 Example

The example file was compiled with cc's **-g** option (which provides symbol table information needed by dbx), and the object file's name was left the default, a.out. Type dbx at the shell command prompt:

```
% dbx
dbx version 2.0 of 5/2/89 0:29.
Type 'help' for help.
enter object file name (default is 'a.out'):
reading symbolic information ...
(dbx)
```

To load a file other than a.out, type the file name on the dbx command line. If the program takes arguments, do *not* type them on the dbx command line; type them after the dbx command.

The example program object file (a.out) is now loaded, and commands can be entered at the dbx prompt, (dbx):

```
(dbx) stop in getline if (st[0]=='q')
[1] stop if st+0*1 = 'q' in getline
(dbx) trace edit in stredit
[2] trace edit in stredit
(dbx) status
[1] stop if st+0*1 = 'q' in getline
[2] trace edit in stredit
(dbx)
```

In the preceding example, dbx is instructed to stop program execution when *st[0]* is 'q' when getline( ) is called. The dbx debugger echoes the command, assigning it the number 1. The next instruction traces the value of the variable *edit* in the

function stredit( ). The command is echoed and assigned the number 2. The **status** command prints the current **stop** and **trace** commands and their numbers. The **run** command starts program execution:

```
(dbx) run
Enter a text line: q w e r t y
stopped in getline at line 79
   79           if (st[i]=='\n')
(dbx) list 70,83
   70   getline (st)
   71       char *st;
   72   {
   73       char c;
   74       int i;
   75
   76       for(i=0; i<=MAX ; i++)
   77       {
   78           st[i]=getchar();
   79           if (st[i]=='\n')
   80               break;
   81       }
   82       st[++i]=' ';
   83   }
(dbx)
```

The characters "q w e r t y" are supplied and execution stops after line 78 because line 78 assigns 'q' to *st[0]*. The **list** command shows source lines and their numbers, in this case lines 70 through 83, which provide context.

The **delete** command deletes any current **stop** or **trace** command:

```
(dbx) status
[1] stop if st+0*1 = 'q' in getline
[2] trace edit in stredit
(dbx) delete 1
(dbx) status
[2] trace edit in stredit
(dbx)
```

To resume running, use the **cont** (continue) command:

```
(dbx) cont

Choose an editing change or combination of changes,
by entering a number or a sum of numbers.
In the case of conflicting changes--for example,
"3" (UPPERCASE and lowercase)--the change with
the lower number will prevail.

    1  UPPERCASE            (highest priority)
    2  lowercase
    4  Initial Capital On All Words
    8  No_blanks
    16 No excess blanks        (lowest priority)

Enter your choice: 9

initially (at line 14 in "crude_editor.c"):     edit = 9
after line 29 in "crude_editor.c":      edit = 1
after line 61 in "crude_editor.c":      edit = 0

QWERTY


execution completed
(dbx)
```

The only command in effect during this run is the trace of the variable *edit*; its initial value is displayed, followed by each change in value. ''QWERTY'' is the program's output.

In the next example, the **trace** is removed, a **stop** is added, and execution is examined source line by source line with the **step** and **next** commands.

The difference between **step** and **next** is that if the next line contains a subroutine call, **step** stops at the beginning of that block, allowing you to step through the subroutine; **next** continues execution until the subroutine returns. To eliminate some typing, aliases are made for both commands:

```
(dbx) status
[2] trace edit in stredit
(dbx) delete 2
(dbx) list 88, 92
   88       int choice;
   89
   90       printf("%s", "Enter a text line: ");
   91       getline(str);  printf("\n\n");
   92
(dbx) stop at 91
[3] stop at 91
(dbx) run
[3] stopped in main at line 91
   91       getline(str);  printf("\n\n");
(dbx) alias s step
(dbx) alias n next
(dbx) n
Enter a text line: 1962   studebaker  hawk


stopped in main at line 93
   93       printf("%s\n", "Choose an editing change or combination of changes,"
);
(dbx) n
Choose an editing change or combination of changes,
stopped in main at line 94
```

```
   94       printf("%s\n", "by entering a number or a sum of numbers.");
(dbx) cont
by entering a number or a sum of numbers.
In the case of conflicting changes--for example,
"3" (UPPERCASE and lowercase)--the change with
the lower number will prevail.

    1  UPPERCASE            (highest priority)
    2  lowercase
    4  Initial Capital On All Words
    8  No_blanks
    16 No excess blanks        (lowest priority)

Enter your choice: 20


1962 Studebaker Hawk


execution completed
(dbx)
```

The **return** command stops execution when program flow returns to the current procedure, or to the procedure supplied as an argument.

In the next example, a **stop** command is used to stop execution immediately so that a **return** command can be issued; a **return** command cannot be issued for a routine until that routine is entered:

```
(dbx) return
process is not active
(dbx) stop if (1==1)
[1] stop if 1 = 1
(dbx) run
stopped in main at line 86
   86   {
(dbx) return
Enter a text line: 1953   hudson   hornet

        .
        .
        .

stopped in . at 0x38
00000038  pushl r0
(dbx) cont
stopped in exit at 0x1e1d
00001e1d  pushl 4(ap)
(dbx) <RETURN>
program exited
(dbx) <RETURN>
can't continue execution
(dbx) <RETURN>
can't continue execution
(dbx)
```

The example above shows execution stopping after getline( ) returns to main( ), and after exit( ) returns to main( ). The example also shows that pressing RETURN with no command executes the last command given.

The **call** command (found only on VAX systems) executes a routine regardless of program flow:

```
(dbx) stop if (1==1)
[2] stop if 1 = 1
(dbx) run
stopped in main at line 86
   86   {
(dbx) call getline("TEST")
stopped in getline at line 76
   76       for(i=0; i<=MAX ; i++)
(dbx) delete 2
(dbx) cont


getline returns successfully
(dbx)
```

8.2.3 Details

The dbx debugger has several other commands, all documented in the *ULTRIX Reference Pages* The **stepi** and **nexti** commands work like **step** and **next**, but they execute a single machine instruction. The **help** command provides a terse list of commands that omits many command options; better online information exists in dbx's reference page. The **quit** command quits the debugger and returns the shell.

The dbx debugger has an optional initialization file, .dbxinit, which contains dbx commands that are read each time dbx is invoked. If there are dbx commands that you issue at the start of every dbx session, place them in .dbxinit and they will be issued automatically.

8.3 error

[*language_processor* |&] **error** [*options*]

8.3.1 Description

The `error` command takes error messages from a compiler or language processor (such as `lint`) and inserts those error messages into the source file at the point the error occurred, thus permitting error messages and source code to be viewed simultaneously without using multiple windows.

The `error` command is usually run with its standard input connected through a pipe to the error message source. Some language processors put error messages on standard output; some put them on standard error. Hence, both should be piped into `error`. The `error` command can handle the error messages produced by the following: `as`, `cc`, `ccom`, `cpp`, `f77`, `ld`, `lint`, `make`, `pc`, and `pi`. If an error message refers to more than one line in a source file, `error` duplicates the message and inserts it before each line.

8.3.2 Example

Apply `lint` to `crude_editor.c` and send the output to `error`:

```
% lint crude_editor.c |& error

        2 non specific errors follow
[lint] printf returns value which is always ignored
[lint] scanf returns value which is always ignored
1 file contains errors "crude_editor.c" (1)

File "crude_editor.c" has 1 error.
        1 of these errors can be inserted into the file.
You touched file(s): "crude_editor.c"
```

Editing `crude_editor.c` and searching for "/*###" reveals the error:

```
getline (st)
    char *st;
{
/*###73 [lint] warning c unused in function getline%%%*/
    char c;
    int i;

    for(i=0; i<=MAX ; i++)
    {
        st[i]=getchar();
        if (st[i]=='0)
            break;
    }
    st[++i]=' ';
}
```

8.3.3 Details

Complete information on `error`'s options can be found in the *ULTRIX Reference Pages*

8.4 gcore

gcore *pid* [,...]

8.4.1 Description

The `gcore` command can help you debug any process on a system. The `gcore` command creates a core image (a ''snapshot'') of each process whose pid (process identification) is supplied. The core image can then be supplied to the `adb` or `dbx` debuggers.

8.4.2 Example

In the following example, `ps` is used to get the pid of `a.out`. This pid is then supplied to `gcore`:

```
% ps
  PID TT STAT  TIME COMMAND
10389 p2 S     0:00 a.out
26297 p2 S     0:00 -sh (csh)
26373 p2 I     0:57 emacs
10393 q7 R     0:00 ps
% gcore 10389
10389: core.10389 dumped
```

The file produced by `gcore` is then used by `adb`:

```
% adb core.10389
```

8.4.3 Details

The process should be stopped before running `gcore` to prevent the process's paging from causing problems. The `gcore` command has no options.

8.5 lint

lint [*options*] *file ...*

8.5.1 Description

The `lint` command checks C source files for code that is wasteful, nonportable, or likely to cause bugs. Run your source files through `lint` before compiling because `lint` is stricter and quicker than most compilers. The `lint` command writes messages to stdout for every error or questionable usage.

8.5.2 Example

The following command runs `lint` on the example program:

```
% lint crude_editor.c
crude_editor.c:
crude_editor.c(73): warning: c unused in function getline
printf returns value which is always ignored
scanf returns value which is always ignored
```

In the preceding example, `lint` displayed three messages. The first message, *c unused in function getline*, calls attention to line 73:

```
getline (st)
    char *st;
{
    char c;             /* This is line 73 */
    int i;

    for(i=0; i<=MAX ; i++)
    {
        st[i]=getchar();
        if (st[i]=='\n')
            break;
    }
    st[++i]='\0';
}
```

The variable c, declared in the function getline( ), is never used; this declaration should be deleted.

Although the grammar is incorrect, messages two and three in the example correctly mention that the return values from printf( ) and scanf( ) are never used. The return values need not be used, but checking the value returned from every function call is considered good programming practice by many, including the creator of `lint`.

8.5.3 Details

The `lint` command has several options to control the types of errors it announces. Some options suppress certain messages, while other options enable certain messages. Complete information on `lint`'s options can be found in the *ULTRIX Reference Pages*

8.6 trace

trace [*options*] *command arguments ...*

8.6.1 Description

The `trace` command traces the system calls made by *command*, and prints the time, pid, call and/or return values and arguments, and puts its output in `trace.dump`.

The `trace` command can help isolate bugs by showing the system calls and their return values immediately before and after a program failure. The `trace` command is also useful for analyzing programs that spend much of their time calling system routines because `prof`, the program that analyzes programs, does not provide system call information.

8.6.2 Example

The following example traces the `ls` command and displays its output:

```
% trace ls -F
#crude_editor.c#        core.10389              ctr.c
a.out*                  core.26373              find.libexc
cc_errors               crude_editor*           temp
cond.c                  crude_editor.c          temp.c
core                    crude_editor_original.c trace.dump
```

A portion of the output from `trace` written to `trace.dump` follows:

```
   ...
096.215546  4728 C execve ( "/bin/ls", 0x100019a0, 0x10001e84 )
   ...
096.313196  4728 C getdirentries ( 5, 0x10013000, 8192/0x2000, 0x1001200c )
096.317102  4728 R getdirentries 512/0x200
096.317102  4728 C lstat ( "#crude_editor.c#", 0x7fffe6fc )
096.317102  4728 R lstat 0
096.317102  4728 C lstat ( "core.10389", 0x7fffe6fc )
096.317102  4728 R lstat 0
096.317102  4728 C old sbreak ( 0x10017ffc )
096.317102  4728 R old sbreak 0
096.321008  4728 C lstat ( "core.26373", 0x7fffe6fc )
096.321008  4728 R lstat 0
   ...
096.336632  4728 C write ( 1, "#crude_editor.c# core.10389  ct", 35 )
096.336632  4728 R write 35
096.403034  4728 C write ( 1, "a.out*   core.26373  find.libex", 33 )
096.403034  4728 R write 33
   ...
096.406940  4728 C close ( 1 )
096.406940  4728 R close 0
096.406940  4728 C close ( 2 )
096.406940  4728 R close 0
096.406940  4728 C exit ( 0 )
   ...
```

8.6.3 Details

The `trace` command has options that allow it to trace groups of processes or certain system calls. Complete information on `trace`'s options can be found in the *ULTRIX Reference Pages*

8.7 RISC Kernel Debugging

This section shows how to debug the ULTRIX kernel, `/vmunix`, on a RISC using various ULTRIX programs. The following subsections show the layouts of the kernel's memory, stacks, and address space; this information should help you understand the debugging procedures.

System Memory Map

Physical Address	KSEG1	Use
0x00030000	0xa0030000 upward	ULTRIX kernel text, data, and bss
0x0002ffff *to* 0x00020000	0xa002ffff 0xa0020000	Additional PROM space (64K)
0x0001ffff *to* 0x0001fc00	0xa001ffff 0xa001fc00	1K netblock (host and client network boot information)
0x0001fbff *to* 0x0001f800	0xa001fbff 0xa001f800	1K ULTRIX save state area
0x0001f7ff *to* 0x0001f400	0xa001f7ff downward 0xa001f400	1K ULTRIX temporary startup stack
0x0001f3ff 0x00010000	0xa001f3ff downward 0xa0010000 upward	dbgmon stack (a few K less than 64K) dbgmon text, data, and bss
0x0000ffff 0x00000500	0xa000ffff downward 0xa0000500 upward	PROM monitor stack PROM monitor bss
0x000004ff *to* 0x00000400	0xa00004ff 0xa0000400	Restart block
0x000003ff *to* 0x00000080	0xa00003ff 0xa0000080	General exception code (note CPU addresses as 0x80000080)
0x0000007f *to* 0x00000000	0xa000007f 0xa0000000	utlbmiss exception code (note CPU addresses as 0x80000000)

Stacks

The kernel has no interrupt stack; only kernel and user stacks. There is an idle stack in ULTRIX V4.0.

Stack	Description
Startup stack	Starts at 0x8001 f7ff, growing downward, and is used during system startup until a kernel stack is available
Kernel stack	Starts at 0xffff e000 (KSEG2 space) and grows down
User struct	Starts at 0xffff c000 (KSEG2 space) and goes up
Per-CPU data base	Starts at 0xffff 8000 (KSEG2 space) and goes up
User stack	Starts at 0x7fff f000 (KUSEG space, one guard page 0x7fff f000 to 7fff ffff) and grows down

Address Space

The system is always in virtual address mode; there is no physical address mode.

Address Space	Description
KSEG0	Not mapped, cached—for kernel text Virtual address: 8000 0000 → 9fff ffff (512 MB)
KSEG1	Not mapped, not cached—for I/O space Virtual address: a000 0000 → bfff ffff (512 MB)
KSEG2	Mapped, cached—for stacks and kernel mallocs Virtual address: c000 0000 → ffff ffff (1 GB)
KUSEG	Mapped, cached—for user space Virtual address: 0 → 7fff ffff (2 GB)

8.7.1 Using nm

For a system crash that gives an EPC (exception PC) on the console, you can use the `nm` command to determine which routine was executing:

```
% nm -n /vmunix
```

The preceding command displays the name list (symbol table) of the vmunix image in numerical order. Find the address that is closest to—but less than—the EPC from the crash; this address is the starting address of the routine executing when the system crashed. Subtract the start address of this routine from the EPC to get the offset from the beginning of the routine in which the error occurred. Then, `dbx` can help you find the offending instruction.

Example of `nm` output:

```
First Kernel text address: 8003,0000 (192k bytes above 8000,0000)
    80030000 T start
```

```
    80030000 T eprol
    800300ac T putstr
    80030148 T lputc
    8003018c T cn_reset
        .
        .
        .
First Kernel data address: is approximately 8011,0000
    80112030 D Sysmap
    8011c830 D Usrptmap
    8011f920 D camap
    8011f930 D kmempt
    8011f930 D ecamap
    80123930 D Forkmap
```

8.7.2 Debugging a RISC Kernel with dbx

To debug a nonrunning kernel, issue the following command:

```
% dbx -k vmunix.n vmcore.n
```

If the system reported an EPC of 0x8000dead when it crashed, dbx can be used to determine where in the kernel that PC is located. The following command decodes nine instructions (and shows line numbers) starting at 0x8000dead. Note that code that is conditioned out (with #ifdef statements) does not count in dbx's line numbering.

```
(dbx) 0x8000dead/9i
8000dead  bleq  8000deaf
8000deaf  cvtfd *-18074(r0),$0.5
8000deb4  movl  (r9),(r6)
8000deb7  decl  8015fe28
8000debd  movl  r7,r1
8000dec0  mfpr  $12,r0
8000dec3  mtpr  r1,$12
8000dec6  ret
8000dec7  halt
(dbx)
```

The following PN dbx commands are useful in kernel debugging:

print gnode[*n*]	Print the gnode struct *n* in the gnode table
print text[*n*]	Print the text struct *n* in the text table
set $pid = *n*	Set process context to process ID *n* (Then you can issue **trace**, **print** *up, **print** *up.u_procp, etc. on that process)
print *up	Print the u_area of the current process
print *up.u_procp	Print the process struct of the current process ID ($pid)

To debug a running kernel, issue the following command:

```
% dbx -k /vmunix
```

8.7.3 Examining Any Process in the System

Issue the following command at the shell to get the pids (process identifications) for every process on the system:

```
% ps -klax vmunix.n vmcore.n
```

The `ps` flags have the following meanings:

-k Use kernel file (`vmcore.n` instead of `/dev/kmem` and `/dev/mem`)

-l Display in long format, giving more information

-a Show all processes (not just your own) associated with a terminal

-x Show processes not associated with a terminal

See the `ps` command in the *ULTRIX Reference Pages* for complete information.

Invoke `dbx` and set *$pid* to the pid of the process you wish to examine; for example, 1125:

```
(dbx) set $pid = 1125
```

Now you can execute **trace**, **print** *up, **print** *up.u_procp, and other commands on process 1125.

The process's stored registers in the u_area are in exception frame format and can be obtained by issuing the following `dbx` command:

```
(dbx) px up.u_ar0[n]
```

8.7.4 Examining the Exception Frame

All error traps and interrupts (except cache parity errors) generate an exception condition. Exception conditions trap to VECTOR(exception) in `locore.s`. The exception routine saves state in the exception frame (on the stack).

For interrupts, VECTOR(VEC_int) is called, which saves additional state on the exception frame, and calls intr( ) (in `trap.c`). The intr( ) routine calls the specific interrupt handler through c0vec_tbl.

For traps, the individual trap routines are called through the causevec. These routines (VEC_addrerr, VEC_ibe, VEC_dbe) in turn call VECTOR(VEC_trap), which saves additional state on the exception frame, and calls trap( ) (in `trap.c`).

A pointer to the exception frame (ep) is passed as an argument to the following routines: trap( ), intr( ), tlbmod( ), tlbmiss( ), and syscall( ). Therefore, by using `dbx` to get a trace, you can find the address of the exception frame (the ep argument). You can then display the exception frame with a `dbx` command such as:

```
(dbx) 0xffffnnnn/41X
```

The offsets within the exception frame are defined as follows (see `/sys/machine/mips/reg.h`):

```
#define EF_ARGSAVE0     0    /* arg save for c calling seq */
#define EF_ARGSAVE1     1    /* arg save for c calling seq */
#define EF_ARGSAVE2     2    /* arg save for c calling seq */
#define EF_ARGSAVE3     3    /* arg save for c calling seq */
#define EF_AT           4    /* r1: assembler temporary */
#define EF_V0           5    /* r2: return value 0 */
```

```
#define EF_V1          6   /* r3:  return value 1 */
#define EF_A0          7   /* r4:  argument 0 */
#define EF_A1          8   /* r5:  argument 1 */
#define EF_A2          9   /* r6:  argument 2 */
#define EF_A3         10   /* r7:  argument 3 */
#define EF_T0         11   /* r8:  caller saved 0 */
#define EF_T1         12   /* r9:  caller saved 1 */
#define EF_T2         13   /* r10: caller saved 2 */
#define EF_T3         14   /* r11: caller saved 3 */
#define EF_T4         15   /* r12: caller saved 4 */
#define EF_T5         16   /* r13: caller saved 5 */
#define EF_T6         17   /* r14: caller saved 6 */
#define EF_T7         18   /* r15: caller saved 7 */
#define EF_S0         19   /* r16: callee saved 0 */
#define EF_S1         20   /* r17: callee saved 1 */
#define EF_S2         21   /* r18: callee saved 2 */
#define EF_S3         22   /* r19: callee saved 3 */
#define EF_S4         23   /* r20: callee saved 4 */
#define EF_S5         24   /* r21: callee saved 5 */
#define EF_S6         25   /* r22: callee saved 6 */
#define EF_S7         26   /* r23: callee saved 7 */
#define EF_T8         27   /* r24: code generator 0 */
#define EF_T9         28   /* r25: code generator 1 */
#define EF_K0         29   /* r26: kernel temporary 0 */
#define EF_K1         30   /* r27: kernel temporary 1 */
#define EF_GP         31   /* r28: global pointer */
#define EF_SP         32   /* r29: stack pointer */
#define EF_S8         33   /* r30: callee saved 8 */
#define EF_RA         34   /* r31: return address */
#define EF_SR         35   /* status register */
#define EF_MDLO       36   /* low mult result */
#define EF_MDHI       37   /* high mult result */
#define EF_BADVADDR   38   /* bad virtual address */
#define EF_CAUSE      39   /* cause register */
#define EF_EPC        40   /* program counter */
```

8.7.5 Examining Stack Frames

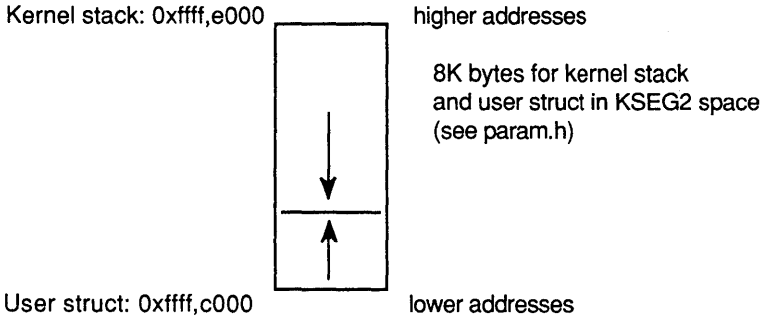
The `odump` utility can be used to create a symbol table dump of `vmunix.n`:

```
% odump -P /vmunix.n > vmunix.syms
```

(See `/usr/include/sym.h`, struct runtime_pdr, for the format of the run-time procedure descriptor created by the loader.)

The fpoff field as shown by `odump` is the frame size for the particular procedure entry. Figure 8-1 illustrates the general format of the stack (stack frames):

Figure 8-1: Stack Frame Layout



Using the symbol table dump, you should be able to work your way back up the call history on the stack. Examples of usage are in `libexc: unwind.c`, `exception.c`, and `exception.h`.

It may be equally productive to start at the top of the kernel stack (high memory) and look for the return address of VEC_syscall on the stack. This return address is where VEC_syscall calls syscall( ), and where the stack frame for entry into syscall( ) has the return address of VEC_syscall saved on the stack.

The following `dbx` command shows the instructions in VEC_syscall, in particular where syscall( ) was called, allowing you to see the return address on the stack:

```
(dbx) VEC_syscall/30i
[VEC_syscall, 0x800c3868]          ori     r5,r16,0x1
[VEC_syscall:590, 0x800c386c]         mtc0    r5,sr
[VEC_syscall:591, 0x800c3870]         sw      r2,20(sp)
[VEC_syscall:592, 0x800c3874]         sw      r3,24(sp)
[VEC_syscall:593, 0x800c3878]         move    r5,r2
[VEC_syscall:594, 0x800c387c]         move    r6,r16
[VEC_syscall:595, 0x800c3880]         jal     syscall
[VEC_syscall:595, 0x800c3884]         nop
[VEC_syscall:596, 0x800c3888]         bne     r2,r0,0x800c3810
[VEC_syscall:596, 0x800c388c]         nop
```

The return address is 0x800c3888. Using `dbx` and the dump of the kernel stack, you can examine the stack to determine what happened to the system.

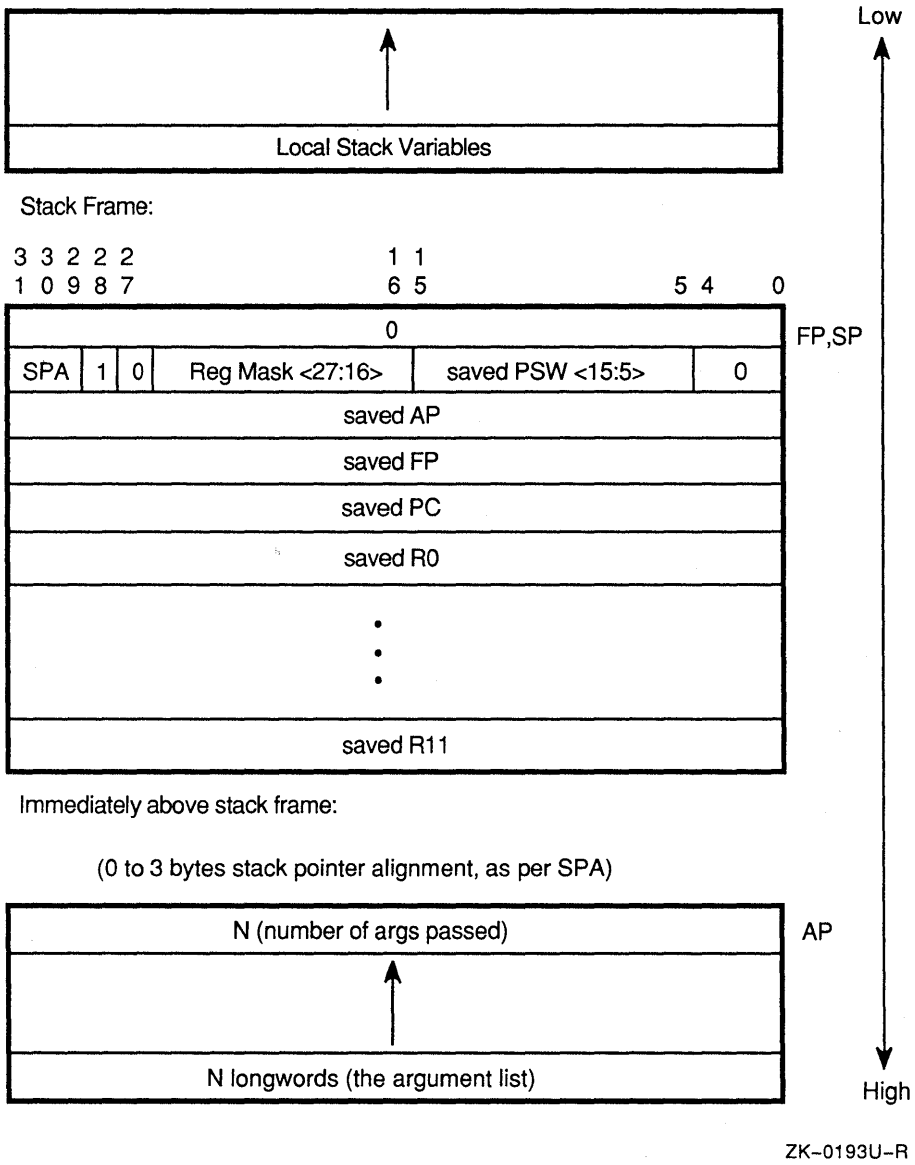
8.7.6 Debugging Hung Systems

Debugging a hung system means finding the real kernel stack. When you force a dump from a hung system, the standard back trace performed by `dbx` is not useful for the currently active process because `dbx` gets the process context out of the u_area, which is old. That is, the u_area contains the process context for the last time the process was context switched.

The kernel stack for each process in the system is located at virtual address 0xffffe000 in KSEG2 space. The system has an array of NPROC u_areas that are 8K bytes each. Each u_area contains the user struct and kernel stack for the process. Even though each user process has its u_area at the same virtual address in KSEG2 space, each u_area is mapped to a unique physical address. When the context

switches, the first two entries in the TLB (safe entries) are established for mapping the u_area for that user process, as illustrated in Figure 8-2.

Figure 8-2: u_area



Within dbx, you can display the kernel stack with a command such as:

```
(dbx) 0xffffd000/1028X
```

The preceding command dumps the kernel stack from low to high memory (most recent events to oldest events).

8.7.7 Forcing a Panic on a System That Is Not Hung

As root, issue the following command:

```
# dbx -k /vmunix /dev/mem
```

The following dbx command forces a panic on the next network interrupt, even in single-user mode (do not issue this command on diskless systems because it will not dump):

```
(dbx) assign ln_softc=0
```

The following command also panics the system:

```
(dbx) assign gnodeops=0
```

Note

Do not overwrite the process structure because dbx will not be able to work on the image. Do not overwrite the console structures because you will not see the panic messages.

8.7.8 Console Commands

The following console commands are useful for debugging a RISC kernel:

dump
>>> **dump -w -x** *address#count*
Dump the contents of memory, starting at *address* and displaying *count* locations of longwords in hex format.

>>> **dump -w -x** *address1*:*address2*
Dump the contents of memory, starting at *address1*, ending at *address2*, displaying longwords in hex format.

>>> **dump -w -x** 0x8001f400:0x8001f800
Dump the startup stack.

examine
>>> **e** [**-b**|**-h**|**-w**] *address*
Examine a byte, halfword, or word at virtual address *address* (To examine physical location 0, use 0x8000 0000.)

go
>>> **go** [*pc*]
Transfer control to given entry point.

help
>>> **help** [*command*]
If no *command* is given, the command menu is displayed.

>>> **?** [*command*]
Same as above.

printenv
>>> **printenv** [*var*]
Display current value of environment variable *var*, or all environment variables.

setenv
>>> **setenv** *var string*
Set the environment variable *var* to be *string*.

unsetenv
>>> **unsetenv** *var*
Delete the environment variable *var*.

test
>>> **t a**
Test all components and subsystems.

booting
>>> **auto**
Use environment variable *bootpath* to boot to multiuser (DS2100 and DS3100 only).

>>> **boot**
Use environment variable *bootpath* to boot single-user on DS2100 and DS3100; boots to multiuser on other systems.

>>> **boot -s**
Boot to single-user (the **-s** option is not available on DS2100 and DS3100).

>>> **boot -f rz(***ctrl, unit,part***)** vmunix
Boot the specified image to single user.

>>> **boot -f mop( )**
Boot from the network to single user.

>>> **boot memlimit=***bytes*
Constrain memory size to *bytes*.

8.7.9 Forcing a Memory Dump on a DS2100 or DS3100

Pressing the restart button halts the machine and clears memory, unless the bootmode is first set to *r* (restart):

```
>>> setenv bootmode r
```

With the bootmode set to *r*, pressing the restart button dumps memory and reboots the machine. The dump may be silent and take several minutes.

8.7.10 Forcing a Memory Dump on a DS5000

Pressing the restart button halts the machine and clears memory, unless the haltaction is first set to *r* (restart):

```
>>> setenv haltaction r
```

With the haltaction set to *r*, pressing the restart button dumps memory and reboots the machine. The dump may be silent and take several minutes.

8.7.11 Forcing a Memory Dump on a DS5400 or DS5800

The break enable switch must be up (pointing to the dot in the circle).

1. Press the break key to get the console prompt.
2. Run the memory dump routine by issuing the **go** command with the kernel start address + 8. In ULTRIX version 3.0 and version 3.1, the kernel start address is 0x80030000; therefore, the command is:

   ```
   >>> go 0x80030008
   ```

8.7.12 Further Information

More information about debugging an ULTRIX kernel on a RISC system can be found in the following header files:

```
/sys/h/proc.h
/sys/h/user.h
/sys/machine/mips/entrypt.h
/sys/machine/mips/frame.h
/sys/machine/mips/pcb.h
/sys/machine/mips/pte.h
/sys/machine/mips/reg.h
```

The `crash` System V program might also be useful.

8.8 VAX Kernel Debugging

This section shows how to debug the ULTRIX kernel, `/vmunix`, on a VAX computer using various ULTRIX programs.

8.8.1 Common Crash Types

This following subsections mention three common crashes and the actions the system takes.

8.8.1.1 Hardware Trap – The system pushes the PSL, PC, code, and trap type onto the interrupt stack. Depending on the trap type, the code is often the last virtual address that was accessed, and is therefore the code that caused the trap (see `/sys/vax/trap.h` for an explanation of trap types). The ULTRIX Trap routine, `/sys/vax/trap.c`, is called through the SCB. Trap in turn calls the panic routine.

An example of a trap is a process that accesses an address outside the process's address space, which causes trap type 8, a segmentation fault.

8.8.1.2 Hardware Machine Check – The system pushes a processor dependent machine check frame onto the interrupt stack. The ULTRIX machine check routine, `/sys/vax/machdep.c`, is called through the SCB. If unrecoverable, the machine check calls the panic routine.

An example of a machine check is a parity memory error.

8.8.1.3 Software Panic – The kernel software detects an internal inconsistency while the system is running on the kernel stack. The kernel routine that detects the inconsistency calls the panic routine (see the *VAX Architecture Handbook* for more information).

8.8.2 Using nm

For a system crash that gives a PC on the console, you can use `nm` to determine which routine was executing:

```
% nm -n /vmunix
```

The preceding command displays the name list (symbol table) of the vmunix image in numerical order. Find the address that is closest to—but less than—the PC from the crash; this address is the starting address of the routine executing when the system crashed. Subtract the start address of this routine from the faulting PC to get the offset from the beginning of the routine in which the error occurred. Then, `adb` can help you find the offending instruction.

8.8.3 Forcing a Crash Dump

If the system is hung, you can force a crash dump. First, halt the processor and enter console mode. Then, issue the following command to get the address of the crash dump routine:

```
>>> E/P/L 4                    ! Get address of dump routine
  P 00000004  00001C00
```

The console's response is the address of the crash dump routine, which can then be run by typing:

```
>>> D PSL 041F0000          ! Set PSL to interrupt stack and IPL to 31
>>> S 80001C00              ! Run the dump routine
```

If the interrupt stack is invalid, the crash dump routine is not called. (The interrupt stack is in kernel address space, starting just below the address of the crash dump routine [*doadump*], and growing down in memory. The interrupt stack has a fixed size of several pages.)

There is another way to force a crash dump. But first, examine the PC and stack pointers, noting their values, because they will be changed by the commands to force a dump:

```
>>> E/G F                   ! Examine general register F (PC)
  G 0000000F  80001EAD
>>> E PSL                   ! Examine the PSL
  M 00000000  04C10004
>>> E SP                    ! Examine the stack pointer
  G 0000000E  000393E8
>>> E/I 0                   ! Examine internal register 0 (KSP)
  I 00000000  7FFFFDAC
>>> E/I 3                   ! Examine internal register 3 (USP)
  I 00000003  7FFFE2F4
>>> E/I 4                   ! Examine internal register 4 (ISP)
  I 00000004  80000C00
```

Now set the PC to -1, and continue:

```
>>> D/G F FFFFFFFF          ! Deposit -1 in PC
>>> D PSL 001F0000          ! Set IPL at 31 to block interrupts
>>> C                       ! Continue processing
```

The preceding commands force a segmentation fault, causing a crash dump. Unfortunately, some machine state is changed using this method. However, all disk writes are completed (as if `sync` had been executed).

If neither of the prior methods work, you may still be able to get a crash dump by initializing the processor before starting the crash dump routine. Initializing sets the processor to a known state, which includes setting the PSL to run on the interrupt stack, setting the IPL to 31, and disabling memory mapping. Unfortunately, even more machine state is changed; depending on the processor, the initialization may corrupt the ISP, KSP, P0BR, P0LR, P1BR, and P1LR.

```
>>> E/P/L 4                 ! Get address of dump routine
  P 00000004  00001C00
>>> I                       ! Initialize the processor
>>> S 80001C00              ! Run dump routine
```

8.8.4 Getting a Stack Trace of any Process

First, get the pid (process identification) of the process to be traced by issuing the `ps` command from the shell:

```
% ps -klax vmunix.n vmcore.n
```

The `ps` flags have the following meanings:

-k Use kernel file (`vmcore.n` instead of `/dev/kmem` and `/dev/mem`)

-l Display in long format, giving more information

-a Show all processes (not just your own) associated with a terminal

-x Show processes not associated with a terminal

See the *ULTRIX Reference Pages* for complete information.

The preceding `ps` command will display the pids of every process on the system. NoteOP the pid of the process you are interested in. Now issue the `/etc/pstat` command with the -p, -a, and -k options:

```
% pstat -pak  vmunix.n vmcore.n
```

The `pstat` flags have the following meanings:

-p Print process table for active processes

-a Describe all process slots

-k Required option when a core file is specified

See the *ULTRIX Reference Pages* for complete information.

Example `pstat` output:

```
195/1044 processes
   LOC    S    F POIP PRI      SIG  UID SLP TIM  CPU  NI   PGRP    PID   PPID
 ADDR   RSS SRSS SIZE    WCHAN     LINK   TEXTP CLKT  TTYP
801e5f70  1    3    0   0         0    0   0 127    0  20      0      0      0
   c96    0    0    0   15f936  1ea170       0          0
801e6030  1    1    0  30         0    0  87 127    0  20      0      1      0
  96df  1c6    0   1f0   1e6030  1e9f30  218e80          0
801e60f0  1    3    0   1         0    0 127 127    0  20      0      2      0
  96bf    0    0  2000   1e60f0       0       0          0
```

Locate the pid you want in the PID field, second from right. The process's location (the memory location of the process structure) is in the LOC field, the leftmost field. (In the preceding example, the location of process 0 is 801e5f70.) Check the process's state and flag codes, second and third fields, labeled S and F.

Invoke `adb` with the following command:

```
% adb -k vmunix.n vmcore.n
```

The following `adb` commands yield the address of the u_area of process 0 (from the preceding example):

```
801e5f70/X     ! Show contents of process structure's first field
801e5f70: 8000feff
     <RETURN> ! Show contents of process structure's second field
801e5f74: 80f03a00
     <RETURN> ! Show contents of process structure's third field
801e5f7c: 81000fff
     <RETURN> ! Show contents of process structure's fourth field
801e5f80: 801ee3e0
     <RETURN> ! Show contents of process structure's fifth field
801e5f88: 80a20fff
```

The fifth field in the process structure contains the address that maps the u_area (see `proc.h`: proc struct and p_addr field); the following `adb` commands set a stack trace for the process:

```
80a20fff$p     ! Set process context for adb
$c             ! Trace stack of process in question
```

8.8.5 adb Command Summary

Either of the following two commands invoke `adb`:

adb -k vmunix vmcore	Invoke `adb` on a crash image
adb -k -w /vmunix /dev/mem	Invoke `adb` on a running system

The following commands are issued within `adb`:

Command	Description
?[*address*] *f*	Print, in format *f*, values in the disk image starting at *address*. Formats for the first three commands: d, Signed decimal word D, Signed decimal longword u, Unsigned decimal word U, Unsigned decimal longword q, Signed octal word Q, Signed octal longword o, Unsigned octal word O, Unsigned octal longword f, Floating point longword F, Floating point double x, Hexadecimal word X, Hexadecimal longword s, String starting at given address
/[*address*] *f*	Print, in format *f*, values in the core file starting at *address*. See ?[*address*] *f* for a list of formats.
= *f*	Print, in format *f*, the virtual address of a symbol. See ?[*address*] *f* for a list of formats.
*(scb-4)$c	Trace stack of whichever stack was currently active (interrupt or kernel) in this format: `func_3 (args) from addr_3` *(newest)* `func_2 (args) from addr_2` `func_1 (args) from addr_1` *(oldest)* func_1 calls func_2 from addr_1 in func_1. Therefore, the stack frame with the saved PC of addr_1 (return address), is the stack frame of func_2
address$c	Trace stack starting from *address* .
routine-name+2[/?]i	Print assembly instructions starting at the beginning of the named routine (+2 skips over the register save mask)
address[/?]i	Print assembly instructions starting at *address*
<RETURN>	Examine the location after the last examined location

Command	Description
^	Examine the location before the last examined location
[/?]w *value*	Write *value* to the last addressed location
$R	Show register contents
range$S	Extend range of symbolic names

8.8.6 adb Scripts

The directory `/usr/lib/adb/` contains `adb` scripts that format kernel data structures. Some sample uses within `adb` follow:

address$<*script*	Apply *script* at *address*
u_block$<u	Apply the user structure script at symbolic address *u_block* (the current u block; that is, the user structure of the current process)
address$<proc	Apply the proc script at *address*, obtained from the user structure
address$<pcb	Apply the pcb script at *address*

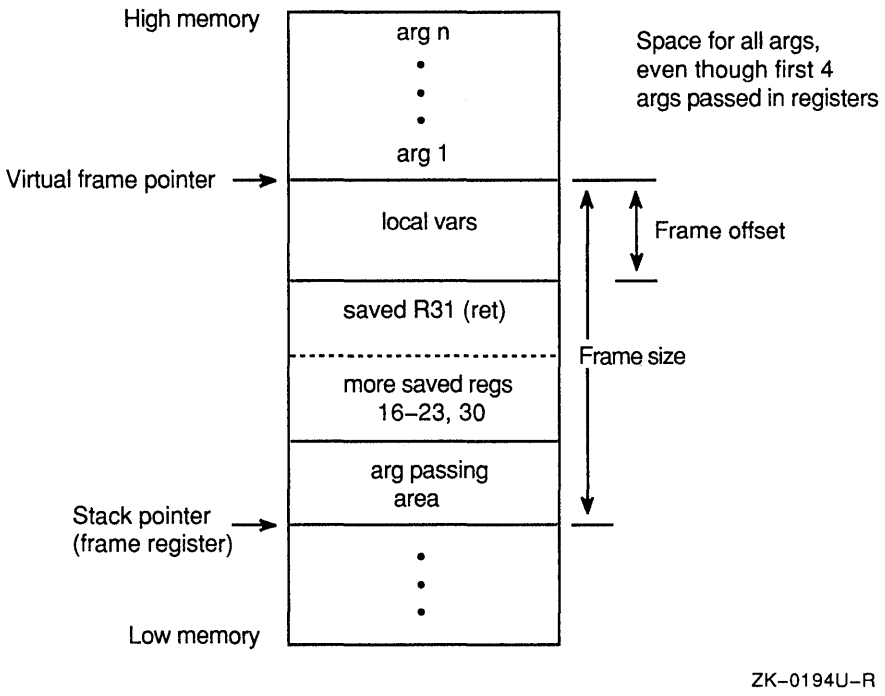
8.8.7 Examining Stack Frames with adb

Using `adb` to examine stack frames is useful for seeing values of local variables. The following are `adb` commands:

(scb-4)/X	scb-4 contains the address of the current stack. If the address is 800*nnnnn*, the system was using the interrupt stack when it crashed; if 7ff*nnnnn*, the system was using the kernel stack
intstack/20X	Starting at the address of *intstack*, print 20 longwords in hex format
u$<u	Show the first item in the user structure, which is the kernel stack pointer (KSP)
KSP/20X	Starting at the address of the kernel stack pointer (KSP), print 20 longwords in hex format

To find a stack frame (a call frame for a procedure call), look for a 0 longword (condition handler) followed by a longword with bit 29 set, which indicates a call (for example, 2e000000). Figure 8-3 illustrates a stack frame in memory.

Figure 8-3: Stack Frame in Memory



The *calls* instruction pushes the argument count onto the stack, then aligns the stack and creates the stack frame (call frame), which is the saved register through the condition handler. (For more information, see the *VAX-11 Architecture Reference Manual.*)

8.8.8 Further Information

The following documents might be useful in debugging an ULTRIX kernel on a VAX computer:

- Bourne, S.R. and Maranzano, J.F. *A Tutorial Introduction to ADB.* In the *ULTRIX Supplementary Documents Volume 2: Programmer*

More information about kernel debugging can be found in the following header files:

```
/sys/h/proc.h
/sys/h/user.h
/sys/vax/pcb.h
/sys/vax/trap.h
```

The `crash` System V program might also be useful.

Programming in a POSIX Environment 9

The ULTRIX system provides a programming environment that allows you to write programs that conform to the following standards:

- IEEE 1003.1–1988 (POSIX) standard
- POSIX Federal Information Processing Standard (FIPS 151–1)
- ISO DIS 9944–1

This chapter refers to all of these standards using the word "POSIX." Your program conforms to the POSIX standard when it includes symbol definitions and library functions that conform to that standard.

The advantage of writing programs that conform to the POSIX standard is that they are easier to port to other platforms. If you write a POSIX-conformant program on another vendor's UNIX-based system, you can move that program to an ULTRIX system with little modification. This ease of porting exists between most systems that supply POSIX-conformant programming environments.

The POSIX environment on ULTRIX consists of the System V shell, POSIX-conformant header files, and a POSIX-conformant function library. You should use the System V shell to write POSIX conformant programs because it contains no extensions to the POSIX standard. Other shells contain features that are extensions to the standard, and you might mistakenly make your program dependent on an extended feature.

To write a POSIX-conformant program, you must use only the POSIX header information and POSIX function library. If you use other header information or `libc` functions instead of standard functions, your program does not conform to the standard.

To include POSIX header information and the POSIX function library, you must compile your program in the POSIX programming environment. This chapter describes how you use the POSIX environment by addressing the following topics:

- Choosing the System V shell
- Using POSIX-conformant header files
- Using the POSIX-conformant function library
- Compiling your program in the POSIX environment
- Correcting errors in the POSIX environment

9.1 Choosing the System V Shell

When you are writing programs that are POSIX-conformant, use the System V shell (`sh5`). This shell's features are implemented to follow the standard.

This section explains how to change your login shell to sh5 and how to modify shell scripts so that they run independently of your login shell.

9.1.1 Using the chsh Command

To change your login shell to the System V shell, use the chsh command. For example, to change the login shell for the login name "Smith," issue the following command:

```
% chsh smith
Changing login shell for smith
shell [/bin/csh]: sh5
%
```

The chsh command modifies your entry in the system password file by changing the login shell field to sh5. (If your system is in a distributed environment, your password file may be distributed from another system. The chsh command does not change distributed password files. Your system administrator must change your entry in the distributed password database. For a description of modifying or adding users to the distributed password database, see the *Guide to System and Network Setup.*)

You must log out and log in again to begin using sh5. The system changes your shell to sh5 when it reads your entry in the password file.

9.1.2 Modifying Shell Scripts

To ensure that your shell scripts run after you change your shell to sh5, add a comment to the first line of each shell script. In the comment, specify what shell the system should use to run that script. For example, in a shell script that runs under csh, add the following comment:

```
#! /bin/csh

date +DATE: %m/%d/%y%nTIME: %H:%M:%S
  .
  .
  .
```

The comment must be the first line in the shell script.

When the system encounters the comment shown in the preceding example, it executes the command specified in the comment. In this case, the system executes csh. The shell receives the commands in the script as csh command arguments; that is, the shell script runs using csh, even though your login shell is sh5.

You should also identify shell scripts that use sh5. The following shows the comment you use to identify shell scripts you write for sh5:

```
#! /bin/sh5

date +DATE: %m/%d/%y%nTIME: %H:%M:%S
  .
  .
  .
```

If you move a shell script you wrote on an ULTRIX system to another UNIX-based system, you might need to modify the comment in the script's first line. Some systems store the System V shell in a file other than /bin/sh5. On those systems, you must modify the comment to name the file that contains the System V shell.

Alternatively, you can create a link to the file that contains the System V shell. For example, suppose a system stores the System V shell in a file named `/bin/sh`. Issue the following command to create a link between that file and `/bin/sh5`:

```
% ln /bin/sh /bin/sh5
```

This `ln` command creates a link that causes the system to execute the `/bin/sh` file when it encounters the `/bin/sh5` pathname.

9.2 Using POSIX Conformant Header Files

The ULTRIX header files contain POSIX-conformant header information. The definitions in the ULTRIX header files are conditional and depend on the definition of the _POSIX_SOURCE preprocessor symbol. When that symbol is defined, POSIX-conformant header information is included in your program. Otherwise, the default ULTRIX header information is included.

You can define the _POSIX_SOURCE symbol in one of three ways: in your program source code, in a local header file, or on the `cc` command line.

To define the symbol in a local header file, create a header file that contains the following line:

```
#define _POSIX_SOURCE
```

Then include the local header file in your source program. For example, if you name the local header file `standard_head`, use the following directive in your source program:

```
# include "standard_head"
```

Be sure to include the local header file in each source file for your program.

You must compile your program in the POSIX programming environment for it to include the POSIX header information. For information on compiling your program in the POSIX environment, see Section 9.4, which also describes how to use a `cc` command line option to define the _POSIX_SOURCE symbol.

9.3 Using the Standard Conformant Function Library

ULTRIX provides a function library that conforms to the POSIX standard. The library is named `libcP`. To use the POSIX library, you must link with that library, in addition to the Berkeley Software Distribution (BSD) `libc`. For a description of linking with the POSIX library, see Section 9.4.

You should be aware of differences between the standard-conformant functions in `libcP` and the functions in BSD `libc`. Table 9-1 lists the functions that differ and explains how the `libcP` functions differ from `libc` functions.

Table 9-1: POSIX Library Functions That Differ from C Library Functions

libcP Function	Difference from the BSD libc Function	Reference Page
`abort`	Closes open files before aborting the process with a SIGABRT signal.	`abort(3)`

Table 9-1: (continued)

libcP Function	Difference from the BSD libc Function	Reference Page
ctermid	Returns a null string if the program has no controlling terminal.	ctermid(3s)
cuserid	Uses the effective user ID, instead of the login user ID.	cuserid(3s)
fclose	Seeks to the byte following the last one your program read or wrote before closing the file.	fclose(3s)
fflush	Writes buffers even if the file is a read-only file.	fclose(3s)
fopen fdopen freopen	Causes the "a" and "a+" mode strings to append with no overwrite.	fopen(3s)
nice	Returns the new priority value minus NZERO. NZERO is the default process priority as defined in limits.h. On ULTRIX systems, NZERO is 20.	nice(3)
opendir	Sets the FD_CLOSEXEC flag on the type *DIR*.	directory(3)
printf fprintf	On success, returns the number of characters printed.	printf(3s)
scanf	Treats the E, G, and X conversion codes the same as the e, g, and x conversion codes.	scanf(3s)
sleep	Can be interupted by signals.	sleep(3)
sprintf	Returns the number of characters formatted. This return value difference affects the syntax of the function call. (See the *ULTRIX Reference Pages* for more information.)	printf(3s)
tzset	Defines the timezone and daylight global variables, which you must declare as *long* and *int*, respectively.	ctime(3)
ungetc	Clears the EOF indicator for the stream.	ungetc(3)

9.4 Compiling in the POSIX Environment

For your program to include POSIX header information and the POSIX function library, you must compile your program in the POSIX environment.

To set your programming environment to POSIX, define the PROG_ENV environment variable, as shown in the following example:

```
% PROG_ENV=POSIX; export PROG_ENV
```

Compile your program using the cc command. For example, if your program consists of three modules named stand_main.c, stand_func_add.c, and stand_func_mult.c, and you have defined the _POSIX_SOURCE symbol and

the PROG_ENV variable, you would issue the following command:

```
% cc stand_main.c stand_func_add.c stand_func_mult.c
```

When you compile a program in the POSIX environment and the _POSIX_SOURCE symbol is defined, the `cpp` preprocessor includes POSIX-conformant header information in your program object. The `ld` linker includes POSIX-conformant functions in your program image.

You can define the _POSIX_SOURCE symbol and the PROG_ENV variable on the `cc` command line by using the `cc` command options `-D` and `-Y`. These options allow you to avoid modifying source code to define the _POSIX_SOURCE symbol and issuing commands to define the PROG_ENV variable. The following example uses `cc` command options to define _POSIX_SOURCE and PROG_ENV:

```
% cc -D_POSIX_SOURCE -YPOSIX stand_main.c stand_func_add.c stand_func_mult.c
```

In this example, the `-D` option defines the _POSIX_SOURCE symbol to the value 1. The `-Y` option sets the programming environment to POSIX. These definitions are in effect only during the execution of the `cc` command. (For more information on the `cc` command, see Chapter 1.)

9.5 Correcting Errors in the POSIX Environment Setup

If your POSIX-conformant programs fail to compile, the problem might be in the setup of your programming environment, instead of in your program code. The problem might be that the POSIX function library is not installed on your system. For information on what is installed on your system and on installing optional software, see your system administrator and the *Advanced Installation Guide.*

Security Guidelines for Programmers 10

Programmers must exercise all the security precautions of a regular user, plus the additional precautions specific to the programming discipline. For example, you must guard passwords, understand the implications of using set user ID (SUID) and set group ID (SGID) programs, and protect your files. Beyond these basic tasks, you also perform the specialized task of designing and writing software programs that others will use repeatedly to handle a variety of tasks.

This chapter presents some general security guidelines for programmers to follow when performing these tasks:

- Passing open file descriptors
- Responding to signals
- Specifying a secure search path
- Protecting permanent and temporary files
- Handling errors
- Writing privileged programs
- Writing SUID and SGID programs
- Authenticating users
- Comparing shell scripts and compiled programs
- Programming in a DECwindows environment

10.1 Passing Open File Descriptors

A child process inherits all the open file descriptors of the parent process.

Passing open file descriptors from one process to another is a security concern because the child has the same type of access to the files that the parent has. You might write an SUID program, for example, that allows users to write data to a sensitive, privileged file. The SUID program could call subprocesses that run in a nonprivileged state. But if the parent SUID process opens a file for writing, the child (or any user running the child process) can write to that sensitive file. Therefore, before creating a subprocess, secure programs close all file descriptors that are not needed by the subprocess.

An efficient way to close file descriptors before executing a new program is to use the `fcntl` call to set the `close-on-exec` flag on the file after you open it. File descriptors that have this flag set are automatically closed each time the program executes a new program. For more information, see the `fcntl`(2) system call in the *ULTRIX Reference Pages*.

10.2 Responding to Signals

The ULTRIX operating system generates signals in response to certain events. The event could be initiated by a user at a terminal (such as quit, interrupt, or stop), by a program error (such as a bus error), or by another program (such as kill).

By default, most signals terminate the receiving process; however, some signals only stop the receiving process. Many signals, such as SIGQUIT or SIGTRAP, write the core image to a file for debugging purposes. A core dump might contain sensitive information, such as passwords.

The `signal` routine, which calls `sigvec`, enables a process to change its response to a signal. This routine enables a process to ignore a signal or call a subroutine when the signal is delivered. However, the SIGKILL and SIGSTOP signals cannot be caught, ignored, or blocked. They are always passed to the receiving process. For more information, see the `sigvec`(2) system call and the `signal`(3) routine in the *ULTRIX Reference Pages.*

Child processes inherit the signal mask that the parent process sets before calling `fork`. The `execve` system call resets all caught signals to the default action; ignored signals remain ignored. Therefore, each process should examine the way it handles signals before executing an `execve` call.

For security reasons, write programs that ignore signals such as quit, interrupt, or stop. A program that implements a background process, for example, should specify `SIG_IGN` for some signals to ignore keystrokes that a user types at the terminal. For more information on the `fork`(2) and `execve`(2) system calls, see the *ULTRIX Reference Pages.*

10.3 Specifying a Secure Search Path

If you use the `popen`, `system`, or `exec*p` routines, which execute `/bin/sh`, be careful when specifying a pathname or defining the shell `path` variable. The `path` variable is a security-sensitive variable because it specifies the search path for executing commands and scripts on your system. For more information on environment variables, see `environ`(7). For more information on the `popen`(3) and `system`(3) routines, see the *ULTRIX Reference Pages.* To specify a secure search path, do the following:

- Specify absolute path names for the `path` variable.
- Do not include public or temporary directories, other users' directories, or the current working directory in your search path. Including these directories increases the possibility of either inadvertently executing the wrong program or with being trapped by a malicious program.
- Make sure that system directories appear before user directories in the list. This prevents you from mistakenly executing a program that might have the same name as a system program.
- Analyze your path syntax. A null entry in a path list specifies the current working directory. In the Bourne and C shells, for example, the colon (:) separates entries in the search list; it should follow (not precede) each entry. For this reason, the first entry following the equal sign (=) should never begin with a colon. Also, no double colons should appear anywhere in the list.

You might want to use the `execve` system call rather than any of the `exec*p` routines because `execve` requires that you specify the pathname. For more information, see the `execve`(2) system call in the *ULTRIX Reference Pages.*

10.4 Protecting Permanent and Temporary Files

If your program uses any permanent files (for example, a database), make sure these files have restrictive permissions and provide controlled access only through your program. These precautions also apply to shared memory segments, semaphores, and IPC mechanisms; set restrictive permissions on all of these objects.

Programs sometimes create temporary files to store data while the program is running. A program should delete these files before it exits. Some security tips for using temporary files follow:

- Programs should not store sensitive information in temporary files, unless the information has been encrypted. Because files are named locations, it is safer to store data in memory than in temporary files.
- Only the owner of the temporary files should have read and write permission on them. It is a good idea to call `umask(077)` at the beginning of the program.
- Temporary files should be created in private directories that are writable only by the owner. If you must use `/tmp`, ask your security administrator to set the sticky bit on the directory (mode 1777), so that files in it can be deleted only by the file owner, the owner of the directory, or the superuser.

A common practice is to create a temporary file, then unlink the file while it is still open. This limits access to any processes that had the file open before the unlink; when the processes exit, the inode is released.

Note that this use of unlink on an NFS-mounted file system takes a slightly different action. The client kernel renames the file and the unlink is sent to NFS only when the process exits. You cannot guarantee that the file will be inaccessible to someone else (although the probability of that happening is minimal), but you can be reasonably sure that the file will be inaccessible when the process exits. In any case, you should always explicitly ensure that no temporary files remain after the process exits.

10.5 Handling Errors

Most system calls and library routines return an integer return code, which indicates the success or failure of the call. Always check the return code to make sure that a routine succeeded. If the call fails, test the global variable `errno` to find out why it failed.

The `errno` variable is set when an error occurs in a system call. You can use this value to obtain a more detailed description of the error condition. This information can help the program decide how to respond, or produce a more helpful diagnostic message. This error code corresponds to an error name in <errno.h>. For more information, see the `errno`(2) system call in the *ULTRIX Reference Pages.* Some `errno` values that indicate a possible security breach are:

EPERM Indicates an attempt by someone other than the owner to modify a file in a way reserved to the file owner or superuser. It can also mean that a user attempted to do something that is reserved for a superuser.

EACCES Indicates an attempt to access a file for which the user does not have permission.

EROFS Indicates an attempt to access a file on a mounted file system when that permission has been revoked.

If your program makes a privileged system call, but the resulting executable program does not have superuser privilege, the program will be compiled, but will fail when it tries to execute the privileged system call. If the security administrator has set up the audit system to log failed attempts to execute privileged system calls, the failure will be audited.

If your program detects a possible security breach, it should not display a diagnostic message that would help an attacker defeat the program. For instance, do not display a message that indicates the program is about to exit because the attacker's real UID did not match a UID in an access file, or even worse, go on to provide the name of the access file. In addition, you could program a small delay before issuing a message to prevent programmed attempts to penetrate your program by systematically trying various inputs.

10.6 Using Privileged Processes

Any process that runs with an effective UID=0 is a privileged process. A program runs with an effective UID=0 if:

- The process executing the program is a superuser process
- The program is SUID `root`

Some system calls and library routines act differently when called by a privileged process. For example, the `setuid` routine sets both the real and effective UIDs, and the `setgid` routine sets both the real and effective GIDs. A nonprivileged process can set these values to only the current real or effective values. A privileged process is not restricted in this fashion and can set these values as it chooses. For more information, see the `setuid`(3) and `setgid`(3) routines in the *ULTRIX Reference Pages.*

Some system calls can only be called from a privileged process. For example, only a privileged process can call `sethostid` or `chroot`. For more information, see the `sethostid`(2) and `chroot`(2) system calls in the *ULTRIX Reference Pages.*

Any calls that use file-system pathnames bypass file protections when called from a privileged process. The following list provides some examples of system calls that act differently for (or can only be called from) a privileged process:

Restricted to root `acct`, `adjtime`, `audcntl`, `audgen`, `chroot`, `exportfs`, `setdomainname`, `sethostid`, `sethostname`, `settimeofday`, `plock`, `reboot`, `setgroups`, `setquota`, `stime`, `swapon`, and `vhangup`.

Different for root `bind`, `chown`, `setpriority`, `setrlimit`, `kill`, `killpg`, `link`, `mknod`, `mount`, `quota`, `setpgrp`, `setregid`, `setreuid`, `setsysinfo`, `socket`, and `ulimit`.

Bypass permissions `msgctl`, `msgsnd`, `msgrcv`, `semctl`, `semop`, `shmctl`, `shmat`, and any calls that use file-system pathnames.

Check the *ULTRIX Reference Pages Section 2: System Calls* for specific information on any calls or routines you plan to use. Make sure that your compile environment

(BSD, SYSTEM_FIVE, or POSIX) does not change the behavior that you expect from a system call or library routine.

10.6.1 Use Minimum Privileges

Because a privileged process has extraordinary powers, create a program that runs as a privileged process only if there is no other way to accomplish the task, and remove superuser privileges (*setuid* not equal to 0) when the process no longer requires them.

Once a privileged process uses the `setreuid` system call to change its real and effective UIDs to something other than 0, it cannot regain superuser privileges. If you write a program that performs both privileged and nonprivileged operations and plan to use `setreuid` to reduce the amount of time the process spends in a privileged state, remember to perform all privileged operations before calling `setreuid`. For more information, see the `setreuid`(2) system call in the *ULTRIX Reference Pages.*

Another approach is to have the program retain superuser privileges and create child processes for nonprivileged operations. Each child process would call `setreuid` to give up its privileged status. This separates privileged from nonprivileged operations within the program, reducing the potential for error or compromise while in a privileged state.

10.6.2 Allocate System Resources with Care

Privileged programs can deliberately or accidentally have a negative effect on the services available to users. For example, privileged programs can call `ulimit` and `nice` to increase file-size limits and set higher priorities for themselves. These changes might have the side effect of denying services to users. Therefore, be careful when you allocate system resources or change system parameters; check for side effects to avoid monopolizing system resources. For more information, see the `ulimit`(2) system call and the `nice`(1) command in the *ULTRIX Reference Pages.*

10.6.3 Know the Process's Real UID

Before performing certain privileged operations, you might want to know who is actually running the program. Use the `getuid` system call to determine the real UID associated with the process. To decide whether or not to allow access to a file, use the `access` system call to determine if the real UID (the user) could access the file in question without the power of a privileged process. You can use this call to decide when to limit the inherent access privileges associated with an effective UID=0. For more information, see the `getuid`(2) and `access`(2) system calls in the *ULTRIX Reference Pages.*

10.6.4 Auditing Security-Relevant Events

If your security administrator has enabled security auditing, the audit daemon, `auditd`, reads data from `/dev/audit` and stores that data in the `/usr/adm/auditlog` file. The audit subsystem can record a wide range of system events. The security administrator can choose events to be logged. For more information on the `auditd`(8) daemon, see the *ULTRIX Reference Pages.* For a complete description of the audit subsystem, see the *ULTRIX Security Guide for Administrators.*

You might want to write a program that generates an audit record for process events. You might also want to change the events that are audited or the items that are recorded in an audit record for a process. Two privileged system calls that enable you to interact with the audit subsystem are:

`audgen` Generates an audit record for a specified operation or event and stores it in the `auditlog`. For more information, see the `audgen`(2) system call in the *ULTRIX Reference Pages.*

`audcntl` Provides control over options offered by the audit subsystem. For more information, see the `audcntl`(2) system call in the *ULTRIX Reference Pages.*

Audit record generation depends on a combination of the system audit mask and the process audit mask. Each process has an audit mask and an audit control flag, both of which originate in `/etc/auth`. Each event that can be audited is represented in both masks. Whether the event is audited depends on the audit control flag, as described in the following list:

- If the process audit control flag is set to **AND**, both masks must indicate that the event should be audited.
- If the process audit control flag is set to **OR**, at least one of the masks must indicate that the event should be audited.
- If the process audit control flag is set to **OFF**, no events for the process are audited.

In Example 10-1, a privileged program uses the `audcntl` system call to turn off auditing for the current process only.

Example 10-1: Using the audcntl Call to Turn off Auditing

```
/* Turns off auditing for this process */
# include <sys/audit.h>
audcntl (SET_PROC_ACNTL, (char *)0, 0, AUDIT_OFF, 0);
```

In Example 10-2, a privileged program uses the `audgen` system call to generate an audit record. The `audgen` call takes three arguments: *event*, *tokenp*, and *argv*. (You can find the lists of event types and token types in `audit.h`.)

The *argv* argument is a pointer to an argument vector. Each entry in the token type array describes the corresponding entry in the argument vector. In this example, `T_CHARP` is a token type describing the "Anything you want to put in the record" string. `T_ERROR` is a token type associated with the error value of -1. You can create an audit record containing up to eight token types and values.

Example 10-2: Using the audgen Call to Generate an Audit Record

```
/* Generates a sample audit record */
# include <sys/audit.h>
    char tmask[AUD_NPARAM];
    struct {
        char *a;
        int b;
    }   aud_arg;
        int i;
   /* Build token mask */
   tmask[0] = T_CHARP;
```

Example 10-2: (continued)

```
tmask[1] = T_ERROR;
tmask[2] = '\0';
/* Fill in values to be recorded */
aud_arg.a = "Anything you want to put in the record";
aud_arg.b = -1;
/* Generate audit record for AUTH_EVENT event */
if (audgen (AUTH_EVENT, tmask, &aud_arg ) == -1 )
     perror ("audgen");
```

In Example 10-3, a privileged program uses the `audcntl` system call to change the events that are audited for this process. This example shows how to adjust the process `audcntl` flag.

The `A_PROCMASK_SET` macro from `audit.h` takes the following four parameters:

- *buf* is the buffer into which the mask is being built.
- The next parameter is the event name, from `syscall.h` for system calls and `audit.h` for events.
- The next parameter is an integer, which indicates whether a successful occurrence of the event should be audited (1 = yes, 0 = no).
- The last parameter is an integer, which indicates if a failed occurrence of the event should be audited (1 = yes, 0 = no).

Example 10-3: Using the audcntl Call to Change the Process Event Mask

```
/* Change the events that are audited for this process */
# include <syscall.h>
# include <sys/audit.h>
# define LEN (SYSCALL_MASK_LEN+TRUSTED_MASK_LEN)
    char buf[LEN];
/* Change process mask to specify auditing for login and failed
 * setgroups (note that 'buf' is initially set to zero). The set
 * process mask is AND'ed with the system mask. This results
 * in only LOGIN and SYS_setgroups being audited for this process
 * (and only if the system mask also specifies LOGIN and/or
 * SYS_setgroups).
 */
if (audcntl (SET_PROC_ACNTL, (char *)0, 0, AUDIT_AND, 0) == -1)
     perror ("audcntl");
A_PROCMASK_SET (buf, SYS_setgroups, 0, 1 );
A_PROCMASK_SET (buf, LOGIN, 1, 1 );
if (audcntl (SET_PROC_AMASK, buf, LEN, 0, 0 ) == -1 )
     perror ("audcntl");
```

10.6.5 Creating Daemons as Privileged Programs

Daemons are long-lived, background processes that provide system-related services. Some standard daemons are the `swapper`, `pagedaemon`, `cron`, `elcsd`, and `lpd` daemons. For more information, see `cron`(8), `elcsd`(8), and `lpd`(8) in the *ULTRIX Reference Pages*. Daemons do not necessarily have to be privileged programs; however, most daemons require privileged access to carry out their tasks. If you create a daemon as a privileged program, take the same care as with any other privileged program.

- Check who is actually requesting the service. Note that this can be a problem if the connection is through a socket, because the information about who is requesting the service is not available from a socket.

  The best approach for safely using sockets in privileged daemons is as follows:

  - Use INET-domain sockets.
  - Have the daemon check that the other side of the connection is a privileged port. A socket can be marked privileged only if the superuser created it. Only privileged sockets can send broadcast packets or bind addresses in privileged portions of an address space. The daemon can determine whether the other side of the connection is a privileged port through the `accept` or `getpeername` system calls. For more information, see the `accept`(2) and `getpeername`(2) system calls in the *ULTRIX Reference Pages*.
  - Write an auxiliary privileged program that connects to the daemon using a privileged port. The auxiliary program can use the `rresvport` routine, for example, to get a privileged port. This requires superuser access. For more information, see the `rresvport`(3) routine in the *ULTRIX Reference Pages*.

  The auxiliary program can perform checks on the user, because it knows who invoked it (either from the audit UID or the real UID). The auxiliary program can then communicate this information to the daemon. The daemon refuses to accept any connection that is not from the auxiliary program.

- Remove the controlling tty using the `ioctl(fd,TIOCNOTTY)` function call.
- Create separate processes for nonprivileged tasks, and remove privileges at the beginning of the routines. If you have separate programs that work with the daemon, in the same way that `lpr` works with `lpd`, make sure that the interaction between the programs cannot be exploited to create a security breach. Put proper protections on both programs. For more information, see `lpr`(1) in the *ULTRIX Reference Pages*.
- Put proper ownership and protections on any permanent or temporary files. Clean up any temporary files before exiting. You might want to use a directory other than `/tmp` depending on the number of files and security issues. Make sure that only the daemon can write to any important directories (or that the sticky bit is set). You might want to create a separate account for the daemon in order to control file ownership and access.

10.7 SUID and SGID Programs

Set user ID (SUID) and set group ID (SGID) programs change the effective UID or GID of a process to the UID or GID of the program. They are a solution to the problem of providing controlled access to system-level files and directories, because they grant a process the access rights of the files' owner.

The potential for security abuse is higher for programs that are either SUID `root` or SGID to any groups that provide write access to system-level files. Simply stated, do not make a program SUID `root` unless there is no other way to accomplish the task. If you must make a program SUID `root`, read the section on privileged processes.

The `chown` system call automatically removes any SUID or SGID bits on a file. This prevents the accidental creation of SUID/SGID programs owned by the `root`

account. For more information, see the chown(2) system call in the *ULTRIX Reference Pages.*

The following list provides suggestions for creating more secure SUID/SGID programs:

- Verify all user-provided pathnames with the access system call.
- Trap all relevant signals to prevent core dumps.
- Test for all error conditions, such as system call return values and buffer overflow.

When possible you should create SGID programs rather than SUID programs. One reason is that file access is generally more restrictive for a group than for a user. If your program is compromised, this reduces the range of actions available to the attacker. Another reason is that it is easier to access files owned by the user executing the SGID program because, when a user executes an SUID program, the original effective UID is no longer available for use for file access. However, when a user executes an SGID program, the user's primary GID is still available as part of the group access list. Therefore, the SGID process still has group access to the files that the primary GID could access.

10.8 Authenticating Users

You need the following to authenticate a user on an ULTRIX system:

- Username
- Password
- /etc/passwd file
- /etc/svc.conf file

If the system is running with enhanced security features enabled, you also must have access to the auth database.

10.8.1 Authenticating a User with Previous Versions of ULTRIX

The method of authenticating a user with previous versions of ULTRIX consisted of the following steps:

1. Use the getpwnam library function to get the passwd database entry corresponding to the username. For information about getpwnam(3), see the *ULTRIX Reference Pages.*
2. Encrypt the password supplied with the crypt function, using the first two characters of the pw_passwd entry in the user's passwd database entry as the salt argument. For information about encrypting passwords and using salt, see crypt(3) in the *ULTRIX Reference Pages.*
3. Compare the output of the crypt function against the string stored in pw_passwd.

If the two encrypted password strings match, the password is valid for the given username. If the username was not found in the passwd database, or if the two encrypted passwords did not match, the authentication fails.

10.8.2 Authenticating a User with the Current Version of ULTRIX

With the current release of ULTRIX, the system administrator may optionally configure the system to store the passwords for each account in a separate database, `auth`, which is not accessible to unprivileged users. In addition to the password, this database contains much additional information about the user, including password expiration information. The contents of the file `/etc/svc.conf` determines if this database is to be used.

The method of authenticating a user now consists of the following steps:

1. Use the `getpwnam` library function to get the `passwd` database entry corresponding to the username. For information about `getpwnam`(3), see the *ULTRIX Reference Pages.*
2. Use the `getsvc` library function to get security information from the `/etc/svc.conf` file. For information about `getsvc`(3), see the *ULTRIX Reference Pages.*
3. Look at the value of the `seclevel` field in the `/etc/svc.conf` file to determine where the password is stored and whether password expiration information is used.
    - If the security level is `SEC_BSD`, the authentication algorithm is the same as that of earlier releases of ULTRIX.
    - If the security level is `SEC_UPGRADE`, password expiration information is used, but the password is taken from the `passwd` database entry unless that password is exactly equal to the string "*". In that case the password is taken from the `auth` database.
    - If the security level is `SEC_ENHANCED`, password expiration information is used and the password is always taken from the `auth` database.
4. Encrypt the password supplied using the first two characters of the old encrypted password as the `salt` argument.
    - If the pasword came from the `passwd` database, use the `crypt` function. For information about encrypting passwords and using `salt`, see `crypt`(3), in the *ULTRIX Reference Pages.*
    - If the password came from the `auth` database, use the `crypt16` function. For information about encrypting passwords with the `crypt16` function and using `salt`, see `crypt`(3), in the *ULTRIX Reference Pages.*

If the two encrypted passwords match, then the password has been verified. However, authentication is not complete until the password expiration information (if applicable) is tested.

This test is performed by checking the password modification time stored in the auth database record against the maximum password lifetime information which is also stored there, using the current system time as a reference. If modification time plus maximum lifetime is less than the current system time, the password has expired and the account is not valid. An additional time factor, called the soft expiration time, may also be used in the calculation to provide a grace period during which users can log into the system provided they change their passwords immediately.

Depending on your application, you may also want to check the A_LOGIN flag in the auth database record for the user.

Note

Although the getpwnam(3) and getauthuid(3) library functions transparently retrieve entries served from remote hosts, you must get a Kerberos ticket-granting ticket before you can obtain auth database entries for hosts served through BIND/Hesiod. See the *Guide to Network Programming* for information about using Kerberos.

The code to implement user authentication for Version 4.0 of ULTRIX is as follows:

```
/*
 * authenticate - a routine to verify a user's password.
 */
#include <pwd.h>
#include <sys/svcinfo.h>
#include <auth.h>
int authenticate(username, passwd)
char *username, *passwd;
{
     struct passwd *pwd, *getpwnam();
     AUTHORIZATION *auth, *getauthuid();
     char *pp, *crypt(), *crypt16(), *(*fp)();
     struct svcinfo *svcinfo;
     auth = (AUTHORIZATION *) 0;
     if(!(pwd=getpwnam(username)))
          return 0; /* no account */
     if(!(svcinfo=getsvc()))
          return 0; /* should never happen */
     switch(svcinfo->svcauth.seclevel) {
     case SEC_BSD:
          pp = pwd->pw_passwd;
          fp = crypt;
          break;
     case SEC_UPGRADE:
          if(!(auth=getauthuid(pwd->pw_uid)))
               return 0; /* no auth entry */
          if(!strcmp(pwd->pw_passwd, "*")) {
               pp = auth->a_password;
               fp = crypt16;
          } else {
               pp = pwd->pw_passwd;
               fp = crypt;
          }
          break;
     case SEC_ENHANCED:
          if(!(auth=getauthuid(pwd->pw_uid)))
               return 0; /* no auth entry */
          pp = auth->a_password;
          fp = crypt16;
          break;
     default:
          return 0; /* bad seclevel in /etc/svc.conf */
     }
     if(!*pp && *passwd)
          return 0; /* bad password */
     if(strcmp((*fp)(passwd, pp), pp))
          return 0; /* bad password */
     if(auth) {
          long expiration, time();
          if(auth->a_pw_maxexp) {
```

```
                expiration = auth->a_pass_mod + auth->a_pw_maxexp;
                if(time((long *) 0) > expiration)
                     return 0; /* password expired */
           }
           if(!(auth->a_authmask & A_LOGIN))
                return 0; /* account disabled */
      }
      return 1;
}
```

10.9 Shell Scripts and Compiled Programs

A shell script is a file containing shell commands. Shell scripts can include variables and flow control constructions. If you must use a shell script to handle sensitive data, set and export `path` before writing the body of the script. Do not make shell scripts SUID or SGID.

Compiled programs enjoy a measure of security that shell scripts do not. You can allow users to execute compiled programs while restricting those users from reading the source files. Because users need both read and execute permission to run a shell script, they have a much better chance of deciphering and compromising the script. For this reason, any program whose compromise represents a security risk should be compiled and made available to the general user only as an executable file.

You should deny access to any source files. Remove read permission for group and other on the executable file to deny users the opportunity to use a debugger on the file.

10.10 Programming in a DECwindows Environment

This section discusses four ways to increase security in a DECwindows programming environment:

- Restrict access control
- Protect keyboard input
- Block keyboard and mouse events
- Protect device-related events

For a detailed description of `Xlib` library calls, see the *Guide to the Xlib Library: C Language Binding*. For a detailed description of the X Window System Protocol, see the *X Window System Protocol: X Version 11*.

10.10.1 Restrict Access Control

The access control list is the key to security in the DECwindows environment. This list names the hosts on the network that can access a workstation display. Users logged in to hosts listed in the access control list can read from, write to, and copy the contents of any window by specifying the window ID. Unlike files, windows cannot be protected from authorized users by setting permissions on them.

When a system is installed, the only host listed in the access control list is the local host. The local host is the system on which the window system is running. For example, when workstation `rook` is booted for the first time, `rook` is the only host listed in its access control list.

The system access control list is stored in a privileged file called `/etc/X*.hosts`. The asterisk specifies the number of the workstation display. When a system is installed, this file is either empty or does not exist. The security administrator maintains this file, usually by leaving it empty to protect the workstations on the network from security attacks. If a user does does not add any hosts to the workstation access control list, using the Security option from the Customize menu, the `/etc/X*.Hosts` file determines the access control list for that workstation.

Table 10-1 lists the `Xlib` library function calls that maintain the access control list:

Table 10-1: Xlib Library Function Calls That Maintain the Access Control List

Call	Purpose
XAddHost	Add a single host to the access control list for a specified workstation display.
XAddHosts	Add the specified hosts to the access control list for a specified workstation display.
XListHosts	List the hosts on the access control list. This call enables a program to find out which hosts can connect to a workstation display.
XRemoveHost	Remove the specified host from the access control list for a specified workstation display.
XRemoveHosts	Remove the specified hosts from the access control list for a specified workstation display.
XEnableAccessControl	Enable the use of the access control list at each workstation.
XDisableAccessControl	Disable the use of the access control list at each workstation.

10.10.2 Protect Keyboard Input

Users logged in to hosts listed in the access control list can call the `XGrabKeyboard` function to take control of the keyboard. When a client has called this function, the X server directs all keyboard events only to that client. Using this call, an attacker could easily grab the input stream from a window and direct it to another window. The attacker could return simulated keystrokes to the window to fool the user running the window. Thus, the user might not realize that anything was wrong.

The ability of an attacker to capture a user's keystrokes threatens the confidentiality of the data stored on the workstation.

DECterm windows provide a secure keyboard mode that directs everything a user types at the workstation keyboard to a single, secure window. Users can set this mode by selecting the Secure Keyboard item from the Commands menu in a DECterm window.

Programs that deal with sensitive data should include a secure keyboard mode. This is especially important if your program prompts a user for a password.

Some guidelines for implementing secure keyboard mode follow:

- Use the `XGrabKeyboard` call to the `Xlib` library.
- Use a visual cue to let the user know that secure keyboard mode has been set, for example, reverse video on the screen.
- Use the `XUngrabKeyboard` function to release the keyboard grab when the user reduces the window to an icon. Releasing the keyboard frees the user to direct keystrokes to another window.

10.10.3 Block Keyboard and Mouse Events

Hosts listed in the access control list can send events to any window if they know its ID. The `XSendEvent` call enables the calling application to send keyboard or mouse events to the specified window. An attacker could use this call to send potentially destructive data to a window. For example, this data could execute the `rm -rf *` command or use a text editor to change the contents of a sensitive file. If the terminal was idle, a user might not notice these commands being executed.

The ability of an attacker to send potentially destructive data to a workstation window threatens the integrity of the data stored on the workstation.

DECterm windows block keyboard and mouse events sent from another client if the `allowSendEvents` resource is set to `false` in the `.Xdefaults` file.

You can write programs that block events sent from other clients. The `XSendEvent` call sends an event to the specified window and sets the `send_event` flag in the event structure to `true`. Test this flag for each keyboard and mouse event that your program accepts. If the flag is set to `false`, the event was initiated by the keyboard and is safe to accept.

10.10.4 Protect Device-Related Events

Device-related events, such as keyboard and mouse events, propagate upward from the source window to ancestor windows until one of the following conditions is met:

- A client selects the event for a window by setting its event mask.
- A client rejects the event by including that event in the `do-not-propagate` mask.

A programmer can use the `XReparentWindow` function to change the parent of a window. This call changes a window's parent to another window on the same screen. All you need to know to change a window's parent is the window ID. With the window ID of the child, you can easily discover the window ID of its parent.

The misuse of the `XReparentWindow` call can threaten security in a windowing system. The new parent window can select any event that the child window does not select.

Take these precautions to protect against this type of abuse:

- A child window should select the events that it needs. This prevents the new parent from intercepting events that propagated upward from the child. Parent windows that centralize event handling for child windows are at greater security risk. An attacker can change the parent and intercept the events intended for the children. Therefore, it is safer for each child window to handle its own events. Events that the child explicitly selects never propagate.
- A child window can specify that events will not propagate further in the window hierarchy. This prevents any event from propagating to the parent

window, regardless of whether the child requested the event.

- A child window can ask to be notified when its parent window is changed by setting the `StructureNotify` or `SubstructureNotify` bit in its event mask. For more information on setting these event masks, see the *Guide to the Xlib Library: C Language Binding*.

10.11 Security Summary

You can increase the security of the programs you write by putting yourself in the place of a potential attacker. The attacker is always looking for a weak link in a system. This chapter points out some weak links a program might create. Be on the lookout for them in the programs that you write.

Here is a quick review:

- Take the following actions after a `fork` call but before an `execve` call:
  - Close all file descriptors, except those needed by the new process.
  - Turn off SUIG/SGID attributes.
  - Specify `SIG_IGN` to ignore signals, such as quit. The quit signal generates a core dump, which might include sensitive information, like passwords.
  - If you invoke a shell, check the syntax of your `path` variable. Use absolute path names and put system directories before user directories. Never specify a public directory. Never include your current working directory or another user's directory in your `path`.
- Avoid storing sensitive files in `/tmp`. If you must use `/tmp`, make sure the sticky bit is set on that directory.
- Check return codes from system calls for attempted security break-ins. If you detect such an attempt, take appropriate action.
- Use the minimum privileges for the minimum amount of time required to do the job.
- In a DECwindows environment, restrict hosts from the access control list. Access control is the key to the security of a workstation's display. Use the `XListHosts` call to `Xlib` to list the hosts that can connect to a workstation display. Use the `XRemoveHost` call to remove a host from that access control list.

System Calls and Library Routines with Security Implications 11

This chapter discusses the ULTRIX system calls and library routines that have security implications for programmers. These calls and routines are listed in alphabetical order. The chapter briefly describes the security relevance of each call and routine, and offers suggestions for enhancing security, where appropriate.

ULTRIX C programs can be compiled for BSD, SYSTEM_FIVE, or POSIX environments. This chapter describes the security relevance of system calls and library routines in the default BSD environment. If you compile a program that executes these calls or routines in either the SYSTEM_FIVE or POSIX environment, there might be some differences. For detailed information, see the *ULTRIX Reference Pages Section 2: System Calls* and the *ULTRIX Reference Pages Section 3: Library Routines.*

Some system calls and library routines that are not covered in this chapter might have implicit security concerns. Also, the misuse of a system call or library routine that does not seem to have any security concerns could threaten the security of a computer system. Ultimately, programmers are responsible for the security implications of their programs.

11.1 System Calls

The following ULTRIX system calls have security relevance for programmers:

access	execve	read	sigsetmask
audcntl*	fcntl	setgroups*	sigvec
audgen*	fork	setpgrp*	stat
chmod*	getgid	setregid*	syscall
chown*	getuid	setreuid*	umask
chroot*	mknod*	sigblock	write
creat	open	sigpause	

* Can only be called by, or acts differently when called by, a privileged process. Also, any calls that use file-system pathnames bypass access permissions when called by a privileged process.

`access` This call checks file access based on the real UID and GID. That is, it tells you if the user could have gained access to a specified object without the privileges gained from your program. For example, a program might need to open files in a directory that the typical user has no permission to read. You need to give the user access to these files under the control of the program, so the program is given the set user ID (SUID) and set group ID (SGID) attributes. However, the user may specify other files, and you need to prevent the user from taking advantage of the program's SUID/SGID powers. The

`access` system call confirms that the user has independent permission to access a specified file.

This call takes two arguments: the file name and the type of access. If the access is allowed, the `access` call returns a 0; if the access is not allowed, the call returns a –1.

`audcntl` If your security administrator has enabled security auditing, this privileged call enables you to control these options offered by the audit subsystem:

- Get or set the system audit mask, which determines the system calls that are audited.
- Get or set the trusted event mask, which determines the trusted events that are audited.
- Get or set the current process's audit mask, which determines (in conjunction with the system audit mask) the system calls that are audited for the current process.
- Get the current process's audit control flags. (See `audit.h`.) The process audit control flag indicates how the process audit mask is applied to the system audit mask to determine the events logged for the process in the `auditlog` file.
- Set the current process's audit control flags. (See `audit.h`.)
- Get or set the system audit switch.
- Flush out the kernel audit buffer to `/dev/audit`, making audit data available to the audit daemon. Data is not normally flushed until the buffer is full. This is useful for making auditing data available immediately.
- Return the audit ID of the calling process, if the process has appropriate privilege.
- Set the audit ID of the calling process. You can pass this option to the audit subsystem only if the audit ID of the calling process is greater than zero, the calling process has appropriate privileges, and the audit ID is not set already.

`audgen` If your security administrator has enabled security auditing, this privileged call generates an audit record for a specified operation or event that is being audited by the current process. The call stores the audit record in the `auditlog` file. The audit record includes standard audit event information, such as identification and timestamp information. For a complete description of the audit subsystem, see the *ULTRIX Security Guide for Administrators*.

The `audgen` call takes three arguments: an integer, indicating the event type of the operation to be audited; a character token type, specifying the category of the auditing information to be stored in the audit record; and a character argument, specifying the information to be stored in the audit record.

You cannot change the values for `audit_id`, `uid`, `ruid`, `pid`, `ppid`, `device`, `IP address`, or `hostid` (secondary tokens for these values are available).

`chmod` This call changes access permissions for a specified file. The `chmod` call has no effect on a process that has already opened a file for reading or writing. This call takes two arguments: the file name and the mode. The following example changes the mode on the file `facts` to 644.

```
chmod("/usr/facts", 0644)
```

`chown` This call changes the owner and group of a specified file (unlike the `chown`(8) command, which changes only the file owner). The `chown` call takes three arguments: file name, owner, and group. Only the superuser can change the owner of a file; however, a group member can change the GID associated with a file. For security reasons, this call clears the SUID and SGID permission bits on the file. Clearing these bits prevents accidental creation of SUID/SGID programs that are owned by `root`.

`chroot` This privileged call changes the `root` directory for an ULTRIX file system. It takes one argument, the address of the pathname of a directory. The `chroot` call sets this directory as the `root` directory, the starting point for pathnames beginning with slash (/).

The `chroot` call is restricted to a privileged process because a user could gain superuser privileges if it were not. For example, a user could create a whole duplicate file system in `/tmp` with hard links to the real files. The only difference is that `/tmp/etc/passwd` would not be linked to `/etc/passwd`. The user then copies `/etc/passwd` to `/tmp/etc/passwd`, changes the root password, uses `chroot` to set root to `/tmp`, and runs `/bin/su`. The `/bin/su` command (really `/tmp/bin/su`) reads `/etc/passwd` (really `/tmp/etc/passwd`) and the user gains superuser privileges.

`creat` This call creates a new file or overwrites an existing one. The `creat` call takes two arguments: file name and access permission. The calling process must have write and execute permission on the directory containing the file.

The file's owner is determined by the effective UID. The file's group is determined by the group of its containing directory. The access permission argument sets the file's permissions. The access permission argument is modified by the file creation mask set by the `umask` shell command or the `umask` system call.

`execve` This call replaces the calling process with the new program.

Note

A group of library routines provide interfaces to the `execve` system call. Each of these routines overlays the calling process with the named file: `execl`, `execv`, `execle`, `execlp`, `execvp`, and `exect`.

The `execve` call passes security-relevant information from the calling process to the newly executed program:

- Real and effective UIDs and GIDs

If the file containing the program to be executed has the SUID or SGID bits set, the effective UID is set to the owner of the new program; the effective GID is set to the group of the new program.

- Open file descriptors (except those with the `close-on-exec` flag set)

  Opening files enables a calling program to communicate with the programs it creates with the `execve` call. However, for security reasons, programs should close all files that are not needed by the new program before calling `execve`. An alternative to closing unneeded files is to use the `fcntl` call to set the `close-on-exec` bits.

- File mode creation mask `(umask)` of the calling process
- Shared memory allocation
- Arguments
- Environment variables

`fcntl` This call enables you to control file and socket descriptors. The `fcntl` call has security relevance because it can lock file regions to prevent more than one process from accessing the same file at the same time. Note that these locks, like `flock`, are only advisory locks; they depend on cooperating processes issuing `fcntl` calls to check lock status before opening files.

File locks are not checked by either the `open` or `access` calls. When used by cooperating processes, `fcntl` protects the integrity of data in the file. In addition, the `fcntl` call can set the `close-on-exec` flag.

A process can set this flag for a file descriptor to close the file before executing a new program with the `execve` system call. A process should close any file that the newly executed program does not need before calling `execve`.

`fork` This call creates a child process that is a copy of the calling process. The child process inherits all security-relevant information from the parent.

`getgid` This call returns the real GID of the current process.

The `getegid` call returns the effective group ID of the current process.

`getuid` This call returns the real UID of the current process.

The `geteuid` call returns the effective UID of the current process.

`mknod` This privileged call creates special files. Nonprivileged users use the `mknod` call to create named pipes. The new file's owner ID is set to the process's effective UID. The new file's group ID is set to the parent directory's GID.

`open` This call opens files. It takes three arguments:

- The file name

- A flag specifying whether the file is to be opened for reading, writing, or both
- If the second argument includes the O_CREAT flag, the access file permission in octal formal

The open call returns a valid file descriptor, or –1 if the open fails (for example, if you try to open a file for reading and the process does not have read permission on the file).

Once a process opens a file, changing permissions on that file and its path does not affect the access of the process.

read This call reads a file opened for reading by the open call. The read call can get information from a file, even if the file's permissions are changed after the file is opened.

setgroups This privileged call sets the group access list of the current process. It takes two arguments: the number of groups and a pointer to an array of integers specifying numeric GIDs.

setpgrp This call determines whether a process receives signals from a terminal. In order for a process to receive a signal from a terminal, the process must have the terminal as its controlling tty and be a member of the terminal's process group. In a BSD environment, only a privileged process can set *pgrp*=0, which blocks all signals from a terminal. Note that you use setpgrp to set a process group from the process end, and ioctl to set a process group from the terminal end.

setregid This call sets the real GID of a process, the effective GID, or both. Only a superuser can modify the real GID associated with a process. Nonprivileged processes can set the effective UID to the real UID. In a BSD environment, nonprivileged processes can set the real GID to either the real or the effective GID.

setreuid This call sets the real UID of a process, the effective UID, or both. Only a superuser can modify the real UID associated with a process. Nonprivileged processes can set the effective UID to the real UID. In a BSD environment, nonprivileged processes can set the real UID to either the real or the effective UID.

sigblock This call adds the specified signals to the set of masked signals.

sigpause This call is similar to the sleep(3) routine in that it enables a process to wait for the arrival of a signal.

sigsetmask This call replaces the current signal mask with a new one.

sigvec This call specifies how a process will handle exceptions or interrupts. It takes two arguments: the number (or name) of a signal, and the routine to call when that signal occurs. Some values for the second argument and their resulting actions follow:

- SIG_IGN ignores the signal.
- SIG_DFL handles the signal in the default manner.
- routine name calls the named routine.

Many security-related programs disable terminal interrupts (such as quit and interrupt) to prevent a user from killing a program from the terminal. For example, the `lock` program that locks a user's terminal uses the `sigvec` call to prevent someone from killing the program by pressing the interrupt or quit key. Many programs trap interrupts so they can delete temporary files before exiting.

Child processes inherit the signals set before the parent process calls `fork`. Therefore, each process should examine the way it handles signals. (The `signal.h` header file lists the system signals.) A process that implements background activities often explicitly specifies `SIG_IGN` for some signals, so that it will not respond to events such as keystrokes.

Privileged processes can send a signal to any existing process. For example, they can use `kill` to kill a process. However, killing a process should be a last resort for removing runaway processes.

`stat` This call returns information about the file path. It takes two arguments: the file name and the address of `stat`, a data structure containing the status information. You do not need read, write, or execute permission of the named file, but you must have search permission on the file path. This status structure is defined in `<sys/stat.h>`.

`syscall` This call performs the system call whose assembly language interface has the specified number, register arguments, and other arguments. It returns the register 0 value of the system call.

`umask` This call sets the process's file mode creation mask and returns the previous value of the mask. The file mode creation mask of a process is inherited by its children. The `umask` call works like the `umask` command. The following example sets the `umask` to 027, which means that only the file owner can write to text files, and only the file owner and group can read those files:

```
umask (027)
```

`write` This call writes data to a file previously opened for writing by the `open` call. Like the `read` call, it is unaffected by changes to the permissions on the file once the file is opened for writing. The SUID/SGID permission bit is turned off after any write to a file, except by `root`.

11.2 Library Routines

Library routines are system services that programs can call. Many library routines use system calls. The following ULTRIX library routines have security implications:

`crypt`	`getauthent`	`getpass`	`putpwent`
`cuserid`	`getgrent`	`getpwent`	`setuid`
`encrypt`	`getlogin`	`popen`	`signal`
`system`	`fopen`		

`crypt` This routine encrypts a password, usually obtained from `getpass`. The `crypt` routine takes two arguments: a character pointer to the typed password (the key), and a two-character string used to jumble the encryption algorithm in `encrypt` (the salt). The salt string can be longer, but only the first two characters are relevant.

One of the advantages of `crypt` is that it uses a significant amount of computer time to encrypt a password. Thus, a cryptanalyst trying to break ULTRIX passwords can spend a lot of time calling `crypt` looking for a match in `/etc/passwd`. However, the encryption method is not tamper proof, and you should not rely on this method alone to protect sensitive files. You must still carefully protect files by setting appropriate protections.

`cuserid` The `cuserid` routine is not a secure or reliable way to learn the indentity of a user because it calls the `getlogin` routine. Instead, use the following routines to return the identity of a user:

```
getpwuid(getuid())
```

`encrypt` This password encryption routine encrypts or decrypts a 64-byte character array of ones and zeroes specified as the first argument (8 bytes of text), depending upon the value of the second argument. If the second argument is zero, the `encrypt` routine encrypts the text in the first argument. If the second argument is one, the `encrypt` routine decrypts the text in the first argument.

`fopen` This routine opens a file; it creates a file, if needed. The `fopen` routine opens a file for reading, writing, or both. This routine has the same security considerations as the `open` system call: once a process opens a file, changing permissions on that file and its containing directories does not affect the original access permissions.

`getauthent` This group of related library routines gets and sets an entry in the authorization database:

The `getauthent` routine returns a pointer to an object containing the fields of a entry in the authorization database.

The `getauthuid` routine looks up the authorization entry for the specified user ID and returns it just as the `getauthent` routine does.

The `storeauthent` routine puts the specified authorization entry in the authorization database, overwriting any existing entry with the same *a_uid* field.

The `setauthent` routine rewinds the authorization database file for subsequent accesses by the `getauthent` routine. It does not affect the operation of the `getauthuid` function, since there is never more than one record per user ID.

The `setauthfile` routine sets the pathname of the file used as the authorization database in all subsequent operations.

The `endauthent` routine closes the authorization database file. Subsequent calls to the `getauthent`, `getauthuid`, and `storeauthent` functions reopen it.

Only a superuser, security administrator, and members of the group `authread` can read information from the authorization database. Only a superuser or the security administrator can modify the authorization database.

`getgrent` This routine reads the next entry in the group file.

The `getgrgid` routine takes a GID as its argument. The routine searches the `/etc/group` file for a matching GID, and returns a pointer to that entry.

The `getgrnam` routine takes a group name as its argument. The routine searches the `/etc/group` file for a matching group name, and returns a pointer to that entry.

The `setgrent` routine resets the group file to its first entry. This has the effect of rewinding the `/etc/group` file.

The `endgrent` routine closes the `/etc/group` file.

`getlogin` This routine does not reliably return the name of the calling user because it can be fooled by I/O redirection to another terminal. The use of `getlogin` is not recommended.

`getpwent` This routine reads the next entry from the `/etc/passwd` file. It opens the file if necessary and then returns the following entry.

The `getpwuid` routine takes a UID as its argument and returns a pointer to the corresponding password file entry. The most reliable way to find out who is running a program is to use the `getuid` routine with `getpwuid`. The `getuid` routine returns the real UID, which is passed to the `getpwuid`, which uses it to look up the user's login name.

The `getpwnam` routine returns a pointer to a `passwd` structure filled with the corresponding password file entry. This structure is defined in `<pwd.h>`.

The `setpwent` routine rewinds the password file.

The `endpwent` routine closes the password file.

`popen` This routine invokes `fork` and `execve` calls, passing as its argument the command specified. It then creates a pipe to the new process using the `pipe` call. Never call this routine from inside a SUID program owned by `root` because the resulting shell would have the superuser privileges. Always specify a complete pathname for the command. Otherwise, another user could alter the pathname to execute a bogus command.

`putpwent` This routine adds or changes an entry in the `/etc/passwd` file.

`setuid` This routine sets both real and effective UID of the current process to the UID specified. The superuser can set real and effective UIDs to any value; other users can set the effective UID to the real UID or the real UID to the effective UID. This means that for nonprivileged users, `setuid` is only useful for SUID programs that toggle between the effective and real UID.

The `setegid` routine sets the effective GID of the current process to the specified GID. The superuser can set the effective GID to any value; other users can set the effective GID only to the real GID.

The `seteuid` routine sets the effective UID of the current process to the specified UID. The superuser can set the effective UID to any value; other users can set the effective UID only to the real UID.

The `setgid` routine sets both the real and the effective GIDs of the current process to the specified GID. The superuser can set the real and effective GID to any value; other users can set the real GID to the effective GID or the effective GID to the real GID.

The `setrgid` routine sets the real GID for the current process to the specified GID. The superuser can set the real GID to any value; other users can set the real GID only to the effective GID.

The `setruid` routine sets the real UID of the current process to the specified UID. The superuser can set the real UID to any value; other users can set the real UID only to the effective UID.

`signal` This routine is a simplified interface to the `sigvec()` system call, which specifies how a process will handle exceptions or interrupts.

`system` This routine issues a shell command. The `system` call runs `/bin/sh` to execute the command specified as its argument. Never call this routine from inside a SUID program owned by `root` because the resulting shell would have the superuser privileges. Always specify a complete pathname for the command. Otherwise, another user could alter the pathname to execute a bogus command.

11.3 Security Summary

The system calls and library routines covered in this chapter can be grouped by category, as shown in the Table 11-1 and Table 11-2.

Table 11-1: Security-Relevant System Calls

Category	System Calls	
File Control	`creat`	`open`
	`fcntl`	`read`
	`mknod`	`write`
Process Control	`fork`	`sigpause`
	`execve`	`sigsetmask`
	`setpgrp`	`sigvec`
	`sigblock`	
File Attributes	`access`	`chroot`
	`chmod`	`stat`
	`chown`	`umask`
User and Group ID	`getegid`	`getuid`
	`getgid`	`setgroups`
	`geteuid`	`setreuid`

Table 11-1: (continued)

Category	System Calls	
Auditing	`audcntl`	`audgen`
General	`syscall`	

Table 11-2: Security-Relevant Library Routines

Category	Library Routines	
File Control	`fopen`	`popen`

Password Handling	`getpass`
	`getpwnam`
	`getpwent`
	`getpwuid`
		`putpwent`
		`setpwent`
		`endpwent`

Process Control	`signal`	

Group Processing	`getgrent`
	`getgrnam`
	`getgrgid`
		`setgrent`
		`endgrent`

Identifying the User	`cuserid` `getlogin`	`getpwuid`
Password Encryption	`crypt`	`encrypt`

User and Group ID	`setuid`
	`setegid`
	`seteuid`
		`setgid`
		`setrgid`
		`setruid`

Authorization	`getauthent`
	`getauthuid`
	`storeauthent`
		`setauthent`
		`setauthfile`
		`endauthent`

C Implementation A

The C language supported by the ULTRIX compiler is an implementation of the language defined in *The C Programming Language* by Kernighan and Ritchie (Prentice Hall, 1978). This appendix discusses the following:

- Specifying `vararg` macros, a requirement for all functions that take a variable number of arguments
- Deviations and extensions to the C language, as defined in *The C Programming Language*
- Translation limits

Specifying the varargs.h Macros

If a function takes a variable number of arguments (for example, the C library functions `printf` and `scanf`), you must use the macros defined in the `varargs.h` header file.

The `va_dcl` macro declares the formal parameter `va_alist`, which is either the format descriptor for the remaining parameters or a parameter itself.

The `va_start` macro must be called within the body of the function whose argument list is to be traversed. The function then can transverse the list or pass its `va_list` pointer to other functions to transverse the list. The type of the `va_start` argument is `va_list`, which is defined in `varargs.h`.

The `va_arg` macro accesses the value of an argument rather than obtaining its address. This macro handles those type names that can be transformed into the appropriate pointer type by appending an asterisk (*), which handles most simple cases.

The argument type in a variable argument list must never be an integer type smaller than `int` and must never be `float`.

For more information, see `varargs`(5) in the *ULTRIX Reference Pages.*

The following example illustrates using `varargs` macros:

```
#include <varargs.h>
#include <stdio.h>
enum operations {load, store, add, sub};
main() {
  void emit();
  emit(load, 'I',  0,  4);
  emit(load, 'I',  4,  4);
  emit(add,  'I');
  emit(store,'I',  0,  4);
}
void
emit(op, va_alist)
/* emit takes a variable number of arguments and prints
```

```
/* them according to the operational format. */
enum operations op;
va_dcl {
va_list arg_ptr;
register int length, offset;
register char type;
va_start(arg_ptr);
switch(op) {
  case add:  /* print operation and length */
    type = va_arg(arg_ptr, int);
    printf("add %c\n", type);
    break;
  case sub:  /* print operation and length */
    type = va_arg(arg_ptr, int);
    printf("sub %c\n", type);
    break;
  case load: /* print operation, offset, and length */
    type = va_arg(arg_ptr, int);
    offset = va_arg(arg_ptr, int);
    length = va_arg(arg_ptr, int);
    printf("load %c %d %d\n", type, offset, length);
    break;
  case store:
    type = va_arg(arg_ptr, int);
    offset = va_arg(arg_ptr, int);
    length = va_arg(arg_ptr, int);
    printf("store %c %d %d\n", type, offset, length);
 }
}
```

The expected output from this code is as follows:

```
load I 0 4
load I 4 4
add I
store I 0 4
```

Deviations

C does not support the `entry` keyword, which has no defined use. Additionally, on the RISC architecture C does not support the `asm` keyword as implemented by some C compilers to allow for the inclusion of assembly language instructions.

Extensions

Extensions to C include the following:

- The `enumeration` type, which is a set of values represented by identifiers called enumeration constants; enumeration constants are specified when the type is defined. For information on the alignment, size, and value ranges of the `enumeration` type, see Chapters 2 and 3.
- The `void` type, which allows you to specify that no value be returned from a function.
- For the RISC architecture, the `volatile` type modifier, which is used when programming I/O devices, and the `signed` type. In addition, the `const` keyword has been reserved for future use. For more information on the `volatile` modifier, see Chapter 2.

- For the VAX architecture, the `const` type modifier. For more information on the `const` modifier, see Chapter 3.

Translation Limits

Table A-1 lists the maximum limits imposed on certain items by the C compiler for the RISC and VAX architectures.

Table A-1: C Compiler Limitations

C Specifications	Maximum
Nesting Levels Compound statements Iterations Selections Conditional compilations	<=30
Maximum number of type modifiers (array, pointers, function, volatile)	9
Case labels	500
Function call parameters	150

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to Languages and Programming
AA-ML94B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

What do you like best about this manual? _____

What do you like least about this manual? _____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line

Reader's Comments

ULTRIX
Guide to Languages and Programming
AA-ML94B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

What do you like best about this manual? _____

What do you like least about this manual? _____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line