AT&T

386 UNIX® System V Release 3.1

Software Development Set Release Notes

AT&T Products and Services

To order documents from the Customer Information Center:

- Within the continental United States, call 1-800-432-6600
- Outside the continental United States, call 1-317-352-8556
- Send mail orders to:

  AT&T Customer Information Center
  Customer Service Representative
  P.O. Box 19901
  Indianapolis, Indiana 46219

To sign up for UNIX system or AT&T computer courses:

- Within the continental United States, call 1-800-221-1647
- Outside the continental United States, call 1-609-639-4458

To contact marketing representatives about AT&T computer hardware products and UNIX software products:

- Within the continental United States, call 1-800-372-2447
- Outside the continental United States, call collect 1-215-266-2973 or 1-215-266-2975

To find out about UNIX system source licenses:

- Within the continental United States, except North Carolina, call 1-800-828-UNIX
- In North Carolina and outside the continental United States, call 1-919-279-3666
- Or write to:

  Software Licensing
  Guilford Center
  P.O. Box 25000
  Greensboro, NC 27420

CONTENTS

386 UNIX SYSTEM V RELEASE 3.1 SOFTWARE DEVELOPMENT SET RELEASE NOTES

Introduction

Overview

These *Release Notes* contain information about the Software Development Set (SDS) package. The SDS package is useful to programmers who:

- Want to develop C language programs
- Do extensive programming in the C language
- Want to enhance the efficiency of a C program written in a UNIX system environment
- Need tools to do advanced programming and symbolic debugging
- Want to work with shared libraries
- Work in an environment where it is necessary to track and maintain versions of files and programs
- Want to optimize and streamline development of interactive, character-oriented, C application programs.

The 386 Software Development Set runs on a computer running 386 UNIX System V Release 3.1.

The SDS software package is made up of two parts as follows:

- C Software Development Set (CSDS)
- Graphics Programming Utilities (GPU).

Conventions Used in This Document

In these *Release Notes,* certain typesetting conventions are followed when command names, command line format, files, and directory names are described. There are also conventions for displays of terminal input and output.

- You must type words that are in **bold** font exactly as they appear. Also, commands, filenames, and directory names appear in **bold**.
- Words in *italics* are variables; you substitute the appropriate values. These values may be filenames or they may be data values.
- CRT or terminal output and examples of source code are presented in `constant-width` font.
- In output and source code examples, a backslash (\) at the end of a line indicates that the line wraps around without a break.
- A command name followed by a number, for example, **prof**(1), refers you to that command's manual page, where the number refers to the section of the manual. These manual pages appear in the *UNIX System V Programmer's Reference Manual* unless otherwise noted.

Contents of the Release

The Software Development Set (SDS) comes in one set of five diskettes (four diskettes for CSDS and one diskette for GPU), the contents of which are displayed in the following table.

Table 1: SDS Utilities

Directory	Files		
/bin (CSDS)	**ar**	**cprs**	**lorder**
	as	**dis**	**make**
	cc	**dump**	**mkshlib**
	chkshlib	**gencc**	**nm**
	conv	**ld**	**size**
	convert	**list**	**strip**
/etc (CSDS)	**install**		
/lib (CSDS)	**basicblk**	**crtn.o**	**mcrt0.o**
	cm4defs	**libc.a**	**mcrt1.o**
	comp	**libc_s.a**	**optim**
	cpp	**libld.a**	**pcrt1.o**
	crt0.o	**libm.a**	**pcrt0.o**
	crt1.o	**libPW.a**	
/usr/add-on/include (GPU)	**chartam.h**	**pbf.h**	**temp.h**
	form.h	**print.h**	**track.h**
	kcodes.h	**subcurses.h**	**wind.h**
	menu.h	**tam.h**	
	message.h	**tamwin.h**	
/usr/add-on/include/sys (GPU)	**font.h**	**mouse.h**	**window.h**
	iohw.h	**signal.h**	

Table 1: SDS Utilities (Continued)

Directory	Files		
/usr/bin	admin	delta	sact
	cb	get	sccsdiff
	cdc	lex	sdb
	cflow	lint	tsort
	comb	lprof	unget
(CSDS)	cscope	m4	val
	ctc	prof	vc
	ctcr	prs	what
	ctrace	regcmp	yacc
	cxref	rmdel	
/usr/bin	captoinfo	infocmp	tic
(GPU)	tput		

Table 1: SDS Utilities (Continued)

Directory	Files		
/usr/include (CSDS)	a.out.h	malloc.h	signal.h
	aouthdr.h	math.h	stand.h
	ar.h	memory.h	stdio.h
	assert.h	mnttab.h	storclass.h
	core.h	mon.h	string.h
	ctype.h	nan.h	stropts.h
	dial.h	nlist.h	strselect.h
	dirent.h	nsaddr.h	syms.h
	errno.h	nserve.h	sys.s
	fatal.h	poll.h	termio.h
	fcntl.h	prof.h	time.h
	filehdr.h	pwd.h	tp_defs.h
	ftw.h	regexp.h	ttysrv.h
	grp.h	reloc.h	unistd.h
	ieeefp.h	rje.h	ustat.h
	ldfcn.h	scnhdr.h	utmp.h
	limits.h	search.h	values.h
	linenum.h	setjmp.h	varargs.h
	macros.h	sgtty.h	
/usr/include (GPU)	curses.h	menu.h	tiuser.h
	eti.h	panel.h	unctrl.h
	form.h	term.h	

Table 1: SDS Utilities (Continued)

Directory	Files		
/usr/lib (CSDS)	basicblk dag flip lib1.a libg.a libmalloc.a libprof.a liby.a	lint1 lint2 llib-lc llib-lc.ln llib-lm llib-lm.ln llib-lmalloc.1	llib-port llib-port.ln lpfx nmf xcpp xpass yaccpar
/usr/lib (GPU)	libcrypt.a libform.a libmenu.a libpanel.a libtam.a libtermcap.a libtermlib.a	llib-lcurses llib-lcurses.a llib-lcurses.ln llib-lform llib-lform.ln llib-lmenu	llib-lmenu.ln llib-lpanel llib-lpanel.ln llib-ltam llib-ltam.ln tamhelp
/usr/lib/ctrace (CSDS)	runtime.c		
/usr/lib/help (CSDS)	ad bd cb cm cmds	co de default ge he	prs rc un ut vc

Table 1: SDS Utilities (Continued)

Directory	Files		
/usr/lib/lex (CSDS)	ncform	nrform	
/usr/lib/libp (CSDS)	libc.a	libm.a	libmalloc.a
/usr/lib/tabset (GPU)	3101 beehive	std teleray	vt100 xerox1720
/usr/options (CSDS)	csoftw.name		
/usr/options (GPU)	graphi.name	terminf.name	
/usr/src/lib/eti/demo (GPU)	form0.c form1.c	form2.c menu0.c	menu1.c

Software Overview

The SDS package has two major parts: the C software development set (CSDS) and the graphics programming utilities (GPU). CSDS can be used for developing, debugging, and improving the efficiency of C language programs. GPU is a set of libraries that promotes fast development of screen management applications. These two parts of the SDS package are discussed in the following subsections.

CSDS

CSDS is a collection of tools and utilities that aid you in:

- Developing C language programs
- Advanced programming, symbolic debugging, and improving C language program efficiency.
- Keeping a history of source code files by recording changes made to these files along with comments on each version.

The C Programming Language Development Tools

The main C programming language development tool is the compiler, and is called by the command **cc**. The other programming development tools discussed in this section are the C preprocessor, optimizer, assembler, link editor, tools for manipulating object files, and libraries.

C Compiler

The C compiler supports the C language as specified in *The C Programming Language*. The significant extensions to the language include the following:

- Arbitrary length names for variables and function names
- Structure assignments and arguments
- Functions returning structure values
- Enumerated data types
- Multiple external variable declarations
- Assembly language escapes from C

- Insertion of arbitrary strings into object modules (useful for version control)
- Floating point support in conformance with the *Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Std 754-1985
- Data type **void**
- Additional preprocessor directives.

cc Command

The **cc** command, the major command of CSDS, calls the C compiler. The **cc** command also controls the other phases of compilation, and, unless programmers use options to specify otherwise, **cc** automatically calls the C preprocessor, assembler, and link editor phases. The command options have many uses, such as suppressing the assembler or link editor or invoking the optimizer. The **cc** command also passes some options to these other programs.

The **cc** command accepts files containing C source code as input. The result of the compilation process is an executable module named **a.out** that reflects the contents of the source files and any referenced library routines. The **cc** command also accepts source files that contain assembly language code as input and passes these files directly to the assembler.

C Preprocessor

The C preprocessor [**cpp**(1)] is automatically called whenever the **cc** command is given C source input. The preprocessor performs file inclusion and macro substitution.

Optimizer

The optimizer, an optional component in the compilation process, improves the efficiency of compiler-generated assembly language code. The optimizer reduces the space requirements and speeds the execution time of the resulting object code.

Assembler and Assembly Language

The assembler [**as**(1)] is available for developing applications that require close interaction with hardware, such as those needed to handle input/output devices and interrupts. The assembler converts assembly language code into a relocatable object module composed of machine code and symbolic information. This component provides assembly language programmers access to predefined macros using the UNIX Operating System **m4** macro processor.

Link Editor

The link editor [**ld**(1)] combines relocatable object modules and libraries to produce either an absolute, executable load module or a relocatable object file for use in further link edits. Executable load modules are in the Common Object File Format (COFF). The link editor performs relocation, resolves external references, and incorporates symbolic debugging information into its output file. It searches libraries to resolve all external references and only loads library routines that define an unresolved external reference.

Tools for Manipulating Object Files

CSDS provides a variety of commands used to read and manipulate object files. Here is a list of some utilities with brief descriptions of their use:

ar	Groups files into a single, portable archive file commonly used as a library
cprs	Compresses object files by removing duplicate structure and union symbolic information
dis	Disassembles object files to allow assembly level debugging
dump	Prints selected parts of the named object files
lorder	Generates an ordered listing of object files for efficient library link editing
nm	Prints the symbolic information in an object file
size	Reports the number of bytes of text, initialized data, and uninitialized data (and their sum) included in an object module
strip	Reduces file storage overhead by removing symbolic information from an object file.

Libraries

CSDS comes with libraries for object files, access to system calls, input/output, string manipulation, mathematical functions, and memory allocation.

Advanced Programming Tools and Utilities

The CSDS package contains an extensive set of tools useful for advanced application programming, debugging, improving the efficiency of your programs, and aiding you in keeping track of the different versions of your programs.

Programming and Debugging Utilities

The programming utilities are specialized utilities helpful in the design and development of application programs and systems. The following list gives a short description of the major programming utilities.

cxref	is a C cross-reference listing generator
ctrace	is a statement-by-statement execution trace facility
cflow	produces a graph of program dependencies
lint	detects faulty and non-portable code
cb	is a C code beautifier
regcmp	compiles regular expressions
mkshlib(1)	makes a shared library. Shared libraries are a feature of UNIX System V Release 3.0, and beyond, that allow several **a.out** files to simultaneously use the same object code.
chkshlib(1)	checks a shared library
sdb(1)	a symbolic debugger used to examine C language executable files and core files and to provide a controlled environment for their execution. When testing C language programs symbolically, breakpoints can be set at executable lines of the source code. These breakpoints force the program to pause at the specified point so that an inspection can be made of the current state of the program.
make(1)	a tool that helps you build and maintain up-to-date versions of programs. **make** simplifies the job of keeping track of which files depend on other files, recently modified files, files that need recompiling after changes, and the sequence of operations needed to make a new version of a program.
lex(1)	a tool that generates programs to be used in simple lexical analysis of text. The **lex** tool reads a file containing specifications of strings to be matched and associated C code. Whenever the lexical analyzer produced by **lex** matches a specified string in its input, it executes the associated C code.
yacc(1)	a tool (Yet Another Compiler-Compiler) that accepts both an LALR(1) grammar specification and associated C code fragments that represent actions to be taken when a found

grammar rule is reduced, and then produces a parser.

All of these utilities are described in the *UNIX System V Programmer's Guide* and the *UNIX System V Programmer's Reference Manual.*

Productivity Utilities

The CSDS package has three utilities that can help an experienced programmer enhance the efficiency of a C program written in a UNIX Operating System environment. These utilities are a browser called **cscope** and two profilers, **lprof** and **prof**.

A browser is an interactive program that helps you examine source files by searching for functions, function calls, macros, and variables that you specify. When it finds them, the browser puts you into an editor at the specified location. Thus, instead of thumbing through a stack of printouts to learn code or locate a bug, you can specify a function or text string and let the browser find it. Then you have the option of examining that portion of code or editing it. Whether you want to familiarize yourself with a program or edit a source file, a browser can help you accomplish your task without your reading the code line by line.

The browser in CSDS, designed for use with C code, is called **cscope**. Programmers responsible for writing programs or maintaining existing programs will be able to edit their source code more efficiently with **cscope**. It is especially helpful for a programmer working on someone else's code.

A profiler is a tool that performs dynamic analysis or analysis of a program at run time; it accomplishes this in two phases. First, the profiler collects data about the code while a program is being executed. Then it displays this data in a readily accessible format. The profiler **lprof** provides line-by-line frequency profiling, reporting how many times each line of source code is executed. To obtain a more representative sample of program performance, you can run a program profiled with **lprof** more than once and then merge the data from the multiple runs. This information can be useful in every stage of software development: designing, prototyping, coding, testing, debugging, and maintenance.

The profiler **lprof** can also be used to determine which lines of source code are executed and how much of the code is exercised. These types of output can be obtained by using the **-x** option and the **-s** option, respectively. These options are convenient for programmers who are interested only in execution coverage and who do not need the additional information that **lprof** normally provides. For example, if you are developing a test suite and want to find out how much code is actually tested by your test suite, run **lprof** with either the **-x** or **-s** option, depending on the level of detail you want.

Another CSDS profiler you may find useful is **prof**. The **prof** profiler reports the amount of time spent in various parts of a program during execution. The use of **prof** is not required for using **lprof**, but by using these profilers together you can increase the efficiency of **lprof**. The **prof** profiler allows you to identify the most time-consuming parts of a program. By running **lprof** on only those parts of code, you can avoid generating uninformative output while targeting sections of code that need pruning. It is therefore recommended that you use **prof** and **lprof** together.

To use these utilities, you must know how to use CSDS in the UNIX system environment. These utilities do not modify code for you; they enable you to find parts of code that deserve further work on your part. For programmers who have not compiled C code or used CSDS before, the basics are covered in the *UNIX System V Programmer's Guide.*

Source Code Control Utilities

A subset of the CSDS utilities, sometimes called the source code control system (SCCS), is specifically designed for source code control. These utilities can be used to record all enhancements and changes to files, along with comments on each version, thus maintaining a history of the changes made. The major SCCS functions include:

- Retrieving any recorded version of a file with comments
- Storing a new version of a file
- Comparing two versions of an SCCS file.

SCCS takes custody of a file, and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file. Each separate set of changes is called a delta. History data can be stored with each version: why the changes were made, who made them, when they were made, etc.

Retrieval of the original or any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification.

Here is a list of SCCS commands.

get	Retrieves versions of SCCS files.
unget	Undoes the effect of a **get -e** prior to the file being delta'd.
delta	Applies deltas (changes) to SCCS files and creates new versions.
admin	Initializes SCCS files, manipulates their descriptive text, and controls delta creation rights.
prs	Prints portions of an SCCS file in user specified format.
sact	Prints information about files that are currently out for edit.
help	Gives explanations of error messages.
rmdel	Removes a delta from an SCCS file. Allows removal of deltas created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any UNIX Operating System file(s) for all occurrences of a special pattern and prints out what follows that pattern. Useful in finding identifying information inserted by the **get** command.
sccsdiff	Shows differences between any two versions of an SCCS file.
comb	Combines consecutive deltas into one to reduce the size of an SCCS file.
val	Validates an SCCS file.
vc	Is a filter that may be used for version control.

For instructions on how to use SCCS and detailed descriptions of SCCS commands, see the "Source Code Control System" Chapter in the *UNIX System V Programmer's Guide.*

Graphics Programming Utilities

GPU is a set of libraries that promote fast development of screen management applications. The GPU libraries are a software tool that enable you to incorporate screen management and data entry capabilities into your programs. GPU contains the following libraries:

- *Curses/Terminfo Low Level Function Library*: This library consists of routines for writing character oriented screen management applications independent of the terminal type. Basic routines are provided for writing to a screen, reading from a screen and building windows. Advanced features are used to change screen attributes, draw line graphics and work with more than one terminal. A major new feature is the incorporation of color. You can specify both the background color for each character and the color of the character itself.

- *High-Level Function Libraries*: The high level function libraries are built on top of curses. They consist of functions that create, manipulate, and display panels, forms, and menus.

  — *Panels*: A panel is a rectangular area containing a curses window that may be displayed in whole or in part on the terminal. Panels provide a depth relationship between curses windows. Panels which are logically below other panels are properly obscured.

  — *Forms*: A form is a multi-page display that contains a set of fields. These fields may be used for data entry, labels, or messages. You can customize the look and behavior of a form or field. The rich set of form commands includes the following: inter-field and intra-field navigation, field editing, data entry, and validation.

  — *Menus*: A menu is a display presenting a collection of items. The end-user can select one or more items and this information is available to the application. You can customize the look and behavior of a menu. Menu commands are provided for item navigation, menu scrolling, and item matching.

- *Terminal Acces Method (TAM) Transition Library*: The TAM Transition Library enables character mode applications developed for the UNIX PC using TAM to run on other processor/terminal configurations. The library maps TAM calls to curses routines.

Software Features

The CSDS package supports character classification and conversion and international date and time formats. The **ctype**(3C), **ctime**(3C) and **cftime**(4) routines have been modified as described in the following subsections. Also, the dynamic tables of the CSDS components **comp** (compiler) and **as** (assembler) are described. Other CSDS features discussed in the following subsections include referencing a shared library from within a shared library, the #hide and #export directives, checking shared library versions with chkshlib(1), and a proposed C language standard.

ctype(3C)

The classification of characters (what constitutes alphabetic, printable, upper or lower case) varies from language to language. The **ctype**(3C) library routines that are used to classify character-coded integer values have been enhanced to recognize other code sets or classifications. Among these is the routine **setchrclass**(3C), which is a new routine used to initialize the character classification and conversion table. It is invoked at program startup and can be invoked directly from users' programs. This means the character set specific table can change dynamically.

ctime(3C)

The **ctime**(3C) routines allow the user to manipulate date and time formats. Several new library functions (**cftime, ascftime**, and an enhanced **tzset**) have been added to **ctime**(3C). These routines support the following features:

- The ability to specify fractional time zones
- The ability to specify start and end dates and times of alternate time zones
- The ability to specify time and date formats with new format field descriptors
- The ability to specify native language translations of month and weekday names.

cftime(4)

The **cftime**(4) manual page describes how to create language specific files. These files contain detailed information such as full and abbreviated month names, full and abbreviated weekday names, and default local time and date formats.

For more information on how to use these features, see **ctime**(3C), **ctype**(3C), **cftime**(4), and **environ**(5) in the *UNIX System V Programmer's Reference Manual.*

Dynamic Tables

Though the C language tends to encourage small functions and source files, some existing applications contained very large source files that failed to compile under previous issues of CSDS because of the fixed size of some tables in the compilation system. In this issue, the tables in the compiler (**/lib/comp**) and the assembler (**/bin/as**) are allocated dynamically.

In the compiler, successful compilation is no longer constrained by the number of symbols, the number of cases in a switch, the number of arguments to a function, etc., except as limited by the amount of memory on your machine. Similarly, the assembler's constraint on the number of symbols has been removed.

Referencing A Shared Library From Within A Shared Library

At times you might need to allow one shared library to directly reference routines in another shared library. One way to do this is with imported symbols. Another way is to reference routines in one shared library from another shared library; use the keyword **noload**, with the **#objects** directive in the shared library specification file. When the **#objects noload** directive is used, the **mkshlib** command will search the libraries listed for unresolved references. You will want to use this feature only when you cannot import symbols explicitly.

The #hide and #export Directives

Two directives, **#hide** and **#export**, can be used in the library specification file to control the visibility of external symbols.

Checking Shared Library Versions with chkshlib(1)

The **chkshlib**(1) command allows you to compare versions of shared libraries to see if they are compatible. This command accepts various combinations of executable files, target shared libraries, and host shared libraries as input and tells you if the library versions are compatible, or if the specified executable could have been built by or can run with the specified host or target shared library.

For more information about shared libraries, see the chapter on shared libraries in the *UNIX System V Programmer's Guide*. The *UNIX System V Programmer's Reference Manual* contains more information about **chkshlib**(1) and **mkshlib**(1).

Proposed Standard for C

As these *Release Notes* were published, no official standard for the C programming language existed. The language accepted by AT&T C compilers follows the definition given in *The C Programming Language* by B. Kernighan and D. Ritchie (Prentice-Hall, 1978). The CSDS package also supports the following extensions.

- Flexnames

  This extension allows variable and function name tokens to be distinct to at least the first 100 characters (rather than the first 8 characters).

- Structure assignments and return values

  This extension allows variables of the same structure type to be assigned to one another. The return value of functions can also be a structure.

- Enumeration types
- Multiple external variable declarations

  This extension makes it possible to have the declaration

  ```
  int i;
  ```

  in multiple source files. All these multiple references resolve to the same address at link edit time.

Currently the X3/J11 task force of the American National Standards Institute (ANSI) is defining a standard for the C language (Draft Proposed American National Standard for Information Systems — Programming Language C, October 1986). The standard proposed by ANSI will allow most current legal C programs to be compiled without any changes. Nevertheless, to ease the possible transition process to the standard, the AT&T C compiler included with CSDS warns about the use of some constructs that may not be legal in the future or may cause portability problems. The following are examples of such constructs.

- Declarations, such as,

  ```
  int  i;
   static int i;
  ```

  produce the warning message

  ```
  warning: i previously declared extern, becomes static.
  ```

- Structure definitions missing semicolons, such as

  ```
  struct x {
    int i
  }
  ```

  produce the warning message

  ```
  warning: syntax requires ; at end of struct/union decl
  ```

Installation Notes

The following text describes the space dependencies and version control as relates to the installation of the SDS package. For complete installation procedures, see the *Operations/System Administration Guide.*

Space Dependencies

The SDS package is installed using the *installpkg*(1) command. The *installpkg*(1) command checks to determine that sufficient free space is available in the **root** and **/usr** file systems. You need approximately 7,600 blocks (512-bite blocks) of memory to install the SDS package.

Version Control

The C software development set portion of the SDS package uses a per file method of version control. If the file being installed already exists on the system and has a release number greater than the file belonging to the package being installed, the existing file will not be overwritten. Files without valid release information are assumed to be older than those belonging to the package being installed.

Software Notes

This section offers some tips on using the SDS package and some software workarounds that enhance the usability of the package.

1. Functions of type *float* or *double* need to be declared in scope whether or not their return values are being used.
2. Elements of type *char* will be sign extended. For zero extension, *unsigned char* must be used.
3. If you are compiling your programs with the -g option enabled so that you can do debugging, it is advisable **NOT** to use the -O option as well. In some cases, the two options invoked jointly will produce multiply defined labels. In addition, you should not use -O when compiling -ql because this in turn turns on the -g flag.
4. The default response to the invalid operation, divide by zero, and overflow exceptions is to take a trap. This behavior may be altered by using the **fpsetmask(3)** function.
5. When an Intel 80287 coprocessor is installed, use of denormalized floating point numbers results in a core dump. The problem is that the 80287 chip does not normalize a denormal number when it is loaded and produces an invalid operation exception when a denormal number is stored to memory. If such problems are encountered, one work-around is to enable the denormalized operand exception and provide a signal handler which normalizes a denormal number. This signal handler must also recognize any other enabled traps (signals).
6. Without an Intel 80287 or 80387 coprocessor installed, the floating point emulator incorrectly returns 0 rather than NaN for any operation on NaN.
7. The IEEE 754 standard for floating point (IEEE Standard for binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985) allows several different methods for detecting overflow. As a result, you should not rely on a particular implementation to signal overflow for a particular operation.

8. Floating point comparisons where one operand is a NaN always result in an invalid operation exception. This is because the Intel 80287 lacks an instruction to make this comparison without getting the exception.
9. **dis(1)** and **sdb(1)** do not recognize the Intel 80387 specific instructions.
10. **pipe(2)** — The documentation states that the maximum number of bytes in a pipe (PIPE_MAX) is defined to be 5120 bytes. The system sets PIPE_MAX to 10240.
11. **ioctl(2)** — The **V_ADDBAD** command (notifies the device drivers of bad sectors) in **ioctl(2)** updates only the table currently in memory and does not update the table on the hard disk. Therefore, all the changes made using **ioctl(2)** with **V_ADDBAD** will be lost when the system is rebooted. Also, if an assigned alternate sector goes bad, there is no way to recover.
12. **ioctl(2)** — The **V_GETPARMS** command in **ioctl(2)** returns the incorrct number of sectors for a 360KB device. The number of sectors reported is 1440; however, the correct value is 720.

Compatibility

This section describes the changes made in this issue of the SDS package that may have an effect on the compatibility of your programs.

The Compiler and cc

The following compatibility notes concern changes made to the CSDS **cc** command or the compiler, **comp**, in this issue of the SDS. These notes apply only if you are porting C programs compiled on an AT&T compilation system (release number less than 4.1) for a different machine.

- The **-B** and **-t** options have been removed from the **cc** command. Previous releases printed a warning message that these options would disappear.
- The handling of aggregate initialization has been changed to conform to the definition given by Kernighan and Ritchie. Initialization where all braces are specified or where only the outermost braces are specified continues to work as before.
- **cc** and **comp** can no longer take the address of a label.

  The following illegal C code will no longer compile:

```
f(){
        int i;
lab:
        i = (int) &lab;
}
```

- Bad structure code, such as the following, is disallowed:

  taking the address of the return value of a function which returns a structure:

```
pst = &(stcall());
```

  using a function return value as an L-value:

```
stcall() = *pst;
```

  taking the address of a structure assignment:

```
pst = &(st1=st2);
```

cpp

The following change was made to **cpp** in this issue of the SDS.

- A missing or invalid macro name in `ifdef`, `ifndef`, `undef`, or `define` is now a fatal error.

  For example:

```
#ifdef 202
#undef
#undef 1abc
```

Changes in C Library Functions

The following list describes changes made to functions in the C library in this issue of the SDS.

ctime(3C)	An **a.out** compiled with previous versions of the **ctime** functions when used with some new legal TZ values will give unexpected results.
ctime(3C)	**ctime** now defaults to GMT if TZ is not set. In previous releases it defaulted to EST.
fgets(3S)	A call to **fgets** on a write-only file returns NULL. In earlier releases, **fgets** always returned the address of the buffer passed to it.
fread(3S), **fwrite**(3S)	The **fread** and **fwrite** functions return zero when **size** is zero or huge. In an earlier release, these two functions always returned **nitems**. **size** and **count** are multiplied to give the number of bytes to be transferred. If the result is larger than the remaining bytes of the file or is not representable within the precision of an integer, fewer items will be read than requested and the number of items actually read will be returned.

scanf(3S) Calls to **scanf** now return EOF on end-of-file. In an earlier release, **scanf** erroneously returned zero.

Environment Variables

The variables CFTIME, CHRCLASS, and LANGUAGE are environment variables in CSDS. Setting them may cause C library functions to change their behavior. Also, the TZ environment variable may be interpreted differently. The following table lists the library functions affected by these variables.

Function	Environment Variables
ctime	TZ
isalnum	CHRCLASS
isalpha	CHRCLASS
iscntrl	CHRCLASS
isdigit	CHRCLASS
isgraph	CHRCLASS
islower	CHRCLASS
isprint	CHRCLASS
ispunct	CHRCLASS
isupper	CHRCLASS
localtime	TZ
tolower	CHRCLASS
toupper	CHRCLASS

The mkshlib Command

Uninitialized external variables (common symbols) are illegal in a shared library. Previously the use of common symbols was discouraged by both the documentation and a **mkshlib** warning message. This warning message is now a fatal error.

Future Directions

This section describes areas of the SDS product that are likely to change in future releases.

1. It is likely that some of the functions defined in **libPW** will be removed in a future release of this product. If you have any code that relies on **libPW**, AT&T recommends that you reimplement it using existing functions in the standard C library or that you retain copies of the **libPW** functions that you need.
2. The **list**(1) command will be removed in a future release of this product.
3. AT&T expects, in a future releases of the SDS, to support the ANSI Standard for the C language once the standard is accepted. That standard introduces the keywords **const**, **signed**, and **volatile**. Programmers should therefore avoid using these words as identifiers in programs.
4. A major feature of the graphics programming utilities (GPU) is the ability to turn on and off any of several video attributes, such as bold, dim, blinking, underlining, reverse video, and others. Future enhancements of GPU will include additional video attributes that enable your programs to use the color capabilities of a wide range of terminals.
5. In keeping with AT&T's ongoing internationalization of the UNIX Operating System, future users will be able to use GPU with keyboards using foreign language character sets, such as Kanga.

Documentation

Essential documentation is provided with the SDS software package when purchased. Additional sets of the Software Development documentation (of which these *Release Notes* are a part) are available and can be ordered. Contact your AT&T Sales Representative/authorized dealer or see the Documentation Roadmap for more details. The Documentation Roadmap can be ordered separately by using the 9-digit number 999-300-427.

Documentation Updates

The following pages represent last minute changes made to the *Programmer's Reference Manual* (*UNIX System V Release 3.1, Version 1*). These change pages should be inserted into the *Programmer's Reference Manual* per the instructions found on the following page.

UNIX SYSTEM V RELEASE 3.1
VERSION 1
PROGRAMMER'S REFERENCE MANUAL
UPDATE TO ISS. 1

This update involves the following actions:

1. **ACTION:** Replace INTRO(1) page 1/INTRO(1) page 2 with the new page.
2. **ACTION:** Replace CC(1) page 2/CC(1) page 3 with the new page.
3. **ACTION:** Replace CHKSHLIB(1) page 2/COMB(1) page 1 with the new page.
4. **ACTION:** Replace COMB(1) page 2/CONV(1) page 1 with the new page.
5. **ACTION:** Replace CPRS(1) page 1/CSCOPE(1) page 1 with the new page.
6. **ACTION:** Replace CSCOPE(1) page 2/CSCOPE(1) page 3 with the new page.
7. **ACTION:** Replace LPROF(1) page 1/LPROF(1) page 2 with the new page.
8. **ACTION:** Replace LPROF(1) page 3/M4(1) page 1 with the new page.
9. **ACTION:** Replace MKSHLIB(1) page 1/MKSHLIB(1) page 2 with the new page.
10. **ACTION:** Replace MKSHLIB(1) page 3/MKSHLIB(1) page 4 with the new page.

11. **ACTION:** Replace LOGINLOG(4) page 1/MDEVICE(4) page 1 with the new page.
12. **ACTION:** Replace MDEVICE(4) page 2/MDEVICE(4) page 3 with the new page.
13. **ACTION:** Replace MTUNE(4) page 1/PASSWD(4) page 1 with the new page.
14. **ACTION:** Replace STUNE(4) page 1/SYMS(4) page 1 with the new page.

NAME

intro – introduction to programming commands

DESCRIPTION

This section describes, in alphabetical order, commands available for your computer. The top of each page indicates the utilities package to which the command belongs. The packages are:

Base System
C Software Development Set
Graphics Programming Utilities

NOTE: The Base System commands (1V) are Form and Menu Language Interpreter (FMLI). They are delivered with the Base System but are typically used by programmers. See the *Programmer's Guide* for more information.

COMMAND SYNTAX

Unless otherwise noted, the commands described accept options and other arguments according to the following syntax:

name [*option*(*s*)] [*cmdarg*(*s*)]

where:

name is the name of an executable file

option is - *noargletter*(*s*) or
- *argletter*<>*optarg*

where:

noargletter is a single letter representing an option without an option-argument

argletter is a single letter representing an option requiring an option-argument

<> is optional white space

optarg is an option-argument (character string) satisfying the preceding *argletter*.

cmdarg is a path name (or other command argument) *not* beginning with "-", or "-" by itself indicating the standard input.

Throughout the manual pages there are references to *TMPDIR, BINDIR, INCDIR, LIBDIR,* and *LLIBDIR*. These represent directory names whose value is specified on each manual page as necessary. For example, *TMPDIR* might refer to **/tmp** or **/usr/tmp**. These are not environment variables and cannot be set. [There is also an environment variable called **TMPDIR** which can be set. See *tmpnam*(3S).]

SEE ALSO

exit(2), wait(2), getopt(3C).

getopts(1) in the *User's/System Administrator's Reference Manual.*

Programmer's Guide.

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of "normal" termination) one supplied by the program [see *wait*(2) and *exit*(2)]. The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and non-zero to indicate troubles such as erroneous parameters, or bad or inaccessible data. It is called variously "exit code", "exit status", or "return code", and is described only where special conventions are involved.

WARNINGS

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

-qp Arrange for profiled code to be produced where the **p** argument produces identical results to the **-p** option [allows profiling with *prof*(1)].

-ql Arrange for code to be produced which will collect line-by-line statement coverage of the program [allows profiling with *lprof*(1)]. This option must be used when compiling a C source file, and when link editing multiple source files.

-E Run only *cpp*(1) on the named C programs, and send the result to the standard output.

-H Print out on *stderr* the path name of each file included during the current compilation.

-O Do compilation phase optimization. This option will not have any effect on **.s** files.

-P Run only *cpp*(1) on the named C programs and leave the result in corresponding files suffixed **.i**. This option is passed to *cpp*(1).

-S Compile and do not assemble the named C programs, and leave the assembler-language output in corresponding files suffixed **.s**.

-V Print the version of the compiler, optimizer, assembler and/or link editor that is invoked.

-Wc,arg1[,arg2...]
Hand off the argument[s] *argi* to pass *c* where *c* is one of [**p02al**] indicating the preprocessor, compiler, optimizer, assembler, or link editor, respectively. For example: **-Wa,-m** passes **-m** to the assembler.

-Y [p02alSILU],*dirname*
Specify a new path name, *dirname*, for the locations of the tools and directories designated in the first argument. **[p02alSILU]** represents:

p	preprocessor
0	compiler
2	optimizer
a	assembler
l	link editor
S	directory containing the start-up routines
I	default include directory searched by *cpp*(1)
L	first default library directory searched by *ld*(1)
U	second default library directory searched by *ld*(1)

If the location of a tool is being specified, then the new path name for the tool will be *dirname/tool*. If more than one **-Y** option is applied to any one tool or directory, then the last occurrence holds.

The *cc* command also recognizes **-C**, **-D**, **-I**, and **-U** and passes these options and their arguments directly to the preprocessor without using the -W option. Similarly, the *cc* command recognizes **-a**, **-l**, **-m**, **-r**, **-s**, **-t**, **-u**, **-x**, **-z**, **-L**, **-M**, and **-V** and passes these options and their arguments directly to the loader. See the manual pages for *cpp*(1) and *ld*(1) for descriptions.

Other arguments are taken to be C compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C compatible routines and are passed directly to the link editor. These programs, together with the results of any compilations specified, are link edited (in the order given) to produce an executable program with name **a.out** unless the **-o** option of the link editor is used.

If the cc command is put in a file *prefix*cc the prefix will be parsed off the command and used to call the tools, i.e., *prefix*tool. For example, OLDcc will call OLDcpp, OLDcomp, OLDoptim, OLDas, and OLDld and will link OLDcrt1.o. Therefore, one MUST be careful when moving the cc command around. The prefix will apply to the preprocessor, compiler, optimizer, assembler, link editor, and the start-up routines.

The C language standard was extended to allow arbitrary length variable names. The option pair "**-Wp,-T -W0,-XT**" will cause cc to truncate arbitrary length variable names.

FILES

file.c	C source file
file.i	preprocessed C source file
file.o	object file
file.s	assembly language file
a.out	link edited output
LIBDIR/*crt1.o	start-up routine
LIBDIR/crtn.o	start-up routine
TMPDIR/*	temporary files
LIBDIR/cpp	preprocessor, *cpp*(1)
LIBDIR/comp	compiler
LIBDIR/optim	optimizer
BINDIR/as	assembler, *as*(1)
BINDIR/ld	link editor, *ld*(1)
LIBDIR/libc.a	standard C library
LIBDIR/libc_s.a	standard C shared library

LIBDIR is usually **/lib**.
BINDIR is usually **/bin**.
TMPDIR is usually **/usr/tmp** but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*() in *tmpnam*(3S)].

SEE ALSO

as(1), ld(1), cpp(1), gencc(1M), lint(1), prof(1), sdb(1), tmpnam(3S).

Kernighan, B. W., and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, 1978.

DIAGNOSTICS

The diagnostics produced by the C compiler are sometimes cryptic.

NOTES

By default, the return value from a compiled C program is completely random. The only two guaranteed ways to return a specific value is to explicitly call *exit*(2) or to leave the function **main**() with a *"return expression;"* construct.

If both input files are target libraries and the **-n** option is set, the output states if the first file references symbols in the second file ("includes" the second).

Compatibility of all other combinations of host shared libraries, target shared libraries, and executable files has no useful meaning and these other combinations of files are not accepted as valid input to *chkshlib*.

SEE ALSO

mkshlib(1).
"Shared Libraries" chapter in the *UNIX System V Programmer's Guide*.

DIAGNOSTICS

Exit status is 0 if no incompatibilities are found, 1 if an incompatibility is found, and 2 if a processing error occurs.

CAVEAT

chkshlib requires that you use the **-i** option whenever you use the **-n** option.

Standard binaries distributed with the UNIX system are stripped and *chkshlib* cannot be used with them.

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [**-o**] [**-s**] [**-p**SID] [**-c**list] files

DESCRIPTION

The *comb* command generates a shell procedure [see *sh*(1)] which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored. The generated shell procedure is written on the standard output.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

-o For each **get -e** generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the **-o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

-s This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

-pSID The SCCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

-clist A *list* [see *get*(1) for the syntax of the *list*] of deltas to be preserved. All other deltas are discarded.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

FILES

s.COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).

help(1), sh(1) in the *User's/System Administrator's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

The *comb* command may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

NAME

conv – common object file converter

SYNOPSIS

conv [**-a**] [**-o**] [**-p**] **-t** target [- ¦ files]

DESCRIPTION

The *conv* command converts object files in the common object file format from their current byte ordering to the byte ordering of the *target* machine. The converted file is written to *file.v*. The *conv* command can be used on either the source (sending) or target (receiving) machine.

Command line options are:

-	Indicates that the names of *files* should be read from the standard input.
-a	If the input file is an archive, produce the output file in the UNIX System V Release 2.0 portable archive format.
-o	If the input file is an archive, produce the output file in the old (pre- UNIX System V) archive format.
-p	If the input file is an archive, produce the output file in the UNIX System V Release 1.0 random access archive format.
-t target	Convert the object file to the byte ordering of the machine (*target*) to which the object file is being shipped. This may be another host or a target machine. Legal values for *target* are: pdp, vax, ibm, x86, b16, n3b, mc68, and m32.

The *conv* command is meant to ease the problems created by a multi-host cross-compilation development environment. The *conv* command is best used within a procedure for shipping object files from one machine to another.

The *conv* command will recognize and produce archive files in three formats: the pre- UNIX System V format, the UNIX System V Release 1.0 random access format, and the UNIX System V Release 2.0 portable ASCII format. By default, *conv* will create the output archive file in the same format as the input file. To produce an output file in a different format than the input file, use the **-a**, **-o**, or **-p** option. If the output archive format is the same as the input format, the archive symbol table will be converted, otherwise the symbol table will be stripped from the archive. The *ar*(1) command with its **-t** and **-s** options must be used on the target machine to recreate the archive symbol table.

EXAMPLE

To ship object files from a VAX computer sytem to a 3B2 computer, execute the following commands:

```
conv –t m32 *.out

uucp *.out.v my3b2!~/rje/
```

NAME

cprs – compress a common object file

SYNOPSIS

cprs [**-p**] file1 file2

DESCRIPTION

The *cprs* command reduces the size of a common object file, *file1*, by removing duplicate structure and union descriptors. The reduced file, *file2*, is produced as output.

The sole option to *cprs* is:

-p Print statistical messages including: total number of tags, total duplicate tags, and total reduction of *file1*.

SEE ALSO

strip(1), a.out(4), syms(4).

NAME

cscope – interactively examine a C program

SYNOPSIS

cscope [**-f** *reffile*] [**-i** *namefile*] [[**-I** *incdir*]] [**-d**] [*files*]

DESCRIPTION

cscope is an interactive screen-oriented tool that helps programmers browse through C source code.

By default, *cscope* examines the C, *yacc,* and *lex* source files in the current directory and builds a symbol cross-reference. It then uses this table to find references to symbols (including C preprocessor symbols), function declarations, and function calls.

cscope builds the symbol cross-reference the first time it is used on the source files for the program being browsed. On a subsequent invocation, *cscope* rebuilds the cross-reference only if a source file has changed or the list of source files is different. When the cross-reference is rebuilt, the data for the unchanged files are copied from the old cross-reference, which makes rebuilding much faster than the initial build.

The following options can appear in any combination:

-f *reffile*
: Use *reffile* as the cross-reference file name instead of the default **cscope.out**.

-i *namefile*
: Get the list of files (file names separated by spaces, tabs, or newlines) to browse from *namefile.* If this option is specified, *cscope* ignores any files appearing on the command line.

-I *incdir*
: Look in *incdir* (before looking in **INCDIR**, the standard place for header files that is normally **/usr/include**) for any **#include** files whose names do not begin with / and that are not specified on the command line or in *namefile* above. (The **#include** files may be specified with either double quotes or angle brackets.) The *incdir* directory is searched in addition to the current directory (which is searched first) and the standard list (which is searched last). If more than one occurrence of **-I** appears, the directories are searched in the order they appear on the command line.

-d Do not update the cross-reference.

Requesting the Initial Search

After the cross-reference is ready *cscope* will display this menu:

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
```

Press the **TAB** key repeatedly to move to the desired input field, type the text to search for, and then press the **RETURN** key.

Issuing Subsequent Requests

If the search is successful, any of these single-character commands can be used:

1-9	Edit the file referenced by the given line number.
SPACE	Display next lines.
+	Display next lines.
-	Display previous lines.
^**e**	Edit all lines.
>	Append the displayed list of lines to a file.

At any time these single-character commands can also be used:

TAB	Move to next input field.
RETURN	Move to next input field.
^**m**	Move to next input field.
^**p**	Move to previous input field.
.	Search with the last text typed.
^**r**	Rebuild the cross-reference.
!	Start an interactive shell (type ^**d** to return to *cscope*).
^**l**	Redraw the screen.
?	Display this list of commands.
^**d**	Exit *cscope*.

Note: If the first character of the text to be searched for matches one of the above commands, escape it by typing a \ (backslash) first.

Substituting New Text for Old Text

After the text to be changed has been typed, *cscope* will prompt for the new text, and then it will display the lines containing the old text. Select the lines to be changed with these single-character commands:

1-9	Mark or unmark the line to be changed.
*	Mark or unmark all displayed lines to be changed.
SPACE	Display next lines.
+	Display next lines.
-	Display previous lines.
a	Mark all lines to be changed.
^**d**	Change the marked lines and exit.
ESCAPE	Exit without changing the marked lines.
!	Start an interactive shell (type ^**d** to return to *cscope*).
^**L**	Redraw the screen.
?	Display this list of commands.

ENVIRONMENT VARIABLES

EDITOR	Preferred editor, which defaults to *vi*(1).
HOME	Home directory, which is automatically set at login.
SHELL	Preferred shell, which defaults to *sh*(1).
TERM	Terminal type, which must be a screen terminal.
VIEWER	Preferred file display program [such as *pg*(1)], which overrides **EDITOR** (see above).

VPATH	An ordered list of directory names, separated by colons. It can be used by *cscope* to search for both source and header files, but the two types of files have different orders of search. If **VPATH** is set, *cscope* searches for source files in the directories specified; if it is not set, *cscope* searches only in the current directory. *cscope* searches for header files in the following order: (1) if **VPATH** is set, in directories specified in **VPATH** and if **VPATH** is not set, in the current directory; (2) in directories specified by the **-I** option (if they exist); and (3) in the standard location for header files (normally **/usr/include**).

FILES

cscope.out	Symbol cross-reference file, which is put in the home directory if it cannot be created in the current directory.
ncscope.out	Temporary file containing new cross-reference before it replaces the old cross-reference.
INCDIR	Standard directory for **#include** files (usually is **/usr/include**).

WARNINGS

cscope recognizes function definitions of the form:

fname blank **(** *args* **)** *white arg_decs white* {

where:

fname	is the function name,
blank	is zero or more spaces or tabs, not including newlines,
args	is any string that does not contain a " or a newline,
white	is zero or more spaces, tabs, or newlines, and
arg_decs	are zero or more argument declarations. *arg_decs* may include comments and white space.

It is not necessary for a function declaration to start at the beginning of a line. The return type may precede the function name; *cscope* will still recognize the declaration. Function definitions that deviate from this form will not be recognized by *cscope*.

NAME

lprof – display line-by-line execution count profile data

SYNOPSIS

lprof [-p] [-s] [-x] [[-I incdir]] [[-r srcfile]] [-c cntfile] [-o prog]
lprof -m file1.cnt file2.cnt [[filen.cnt]] [**-T**] **-d** destfile.cnt

DESCRIPTION

lprof is a tool for dynamic analysis; that is, the analysis of a program at run time. Specifically, *lprof* identifies the most frequently executed parts of source code and parts of code that are never executed.

lprof interprets a profile file (**prog.cnt** by default) produced by the profiled program *prog* (*a.out* by default) that has been compiled with the **-ql** option of *cc*(1). This *cc* command option arranges for code to be inserted to record run-time behavior and for data to be written to a file at the end of execution.

By default, *lprof* prints a listing of source files (the names of which are stored in the symbol table of the executable file), each line preceded by its line number (in the file) and the number of times it was executed.

The following options may appear singly or be combined in any order:

-p Print listing, each line preceded by the line number and the number of times it was executed (default). This option can be used together with the **-s** option to print both the source listing and summary information.

-s Print summary information of percentage of lines of code executed per function.

-x Instead of printing the execution count numbers for each line, print each line preceded by its line number and a [U] if the line was not executed. If the line was executed, print only the line number.

-I *incdir*
Look for source or header files in the directory **incdir** in addition to the current directory and the standard place for **#include** files (usually **/usr/include**). You can specify more than one directory with **-I** on one command line.

-r *srcfile*
Instead of printing all source files, print only those files named in **-r** options (to be used with the **-p** option only). You can specify multiple files with **-r** on one command line.

-c *cntfile*
Use the file *cntfile* instead of **prog.cnt** as the input profile file.

-o *prog* Use the name of the program *prog* instead of the name used when creating the profile file. Because the program name stored in the profile file contains the relative path, this option is necessary if the executable file or profile file has been moved.

Merging Data Files

lprof can also be used to merge data files. The **-m** option must be accompanied with the **-d** option:

-m *file1.cnt file2.cnt* [*filen.cnt*] **-d** *destfile.cnt*
: Merge the data files **file1.cnt** through **file*n*.cnt** by summing the execution counts per line, so that data from several runs can be accumulated. The result is written to **destfile.cnt**. The data files must contain profiling data for the same **prog** (see the **-T** option below).

-T
: Time stamp override. Normally, the time stamps of the executable files being profiled are checked, and data files will not be merged if the time stamps do not match. If **-T** is specified, this check is skipped.

Controlling the Run Time Profiling Environment

The environment variable **PROFOPTS** provides run time control over profiling. When a profiled program is about to terminate, it examines the value of **PROFOPTS** to determine how the profiling data is to be handled.

The environment variable **PROFOPTS** is a comma-separated list of options interpreted by the program being profiled. If **PROFOPTS** is not defined in the environment, then the default action is taken: the profiling data is saved in a file (with the default name, **prog.cnt**) in the current directory. If **PROFOPTS** is set to the null string, no profiling data is saved. The following are the available options:

msg=[y|n]
: If **msg=y** is specified, a message stating that profile data is being saved is printed to *stderr*. If **msg=n** is specified, print only profiling error messages. The default is **msg=y**.

merge=[y|n]
: If **merge=n** is specified, do not merge data files after successive runs. The data file is overwritten after each execution. If **merge=y** is specified, the data will be merged. The merge will fail if the program has been recompiled; the data file will be left in **TMPDIR**. The default is **merge=n**.

pid=[y|n]
: If **pid=y** is specified, the name of the data file will include the process ID of the profiled program. This allows the creation of different data files for programs calling *fork*(2). If **pid=n** is specified, the default name is used. The default is **pid=n**.

dir=*dirname*
: Place the data file in the directory **dirname** if this option is specified. Otherwise, the data file is created in the directory that is current at the end of execution.

file=*filename*
: Use **filename** as the name of the data file in **dir** created by the profiled program if this option is specified. Otherwise, the default name is used.

FILES

prog.cnt	for profile data
*TMPDIR/**	temporary files

TMPDIR is usually **/usr/tmp**, but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*( ) in *tmpnam*(3S)].

SEE ALSO

cc(1), prof(1), fork(2), tmpnam(3S).

WARNINGS

For the **-m** option, if **destfile.cnt** exists, its previous contents are destroyed.

Optimizing functions may result in the loss of some line number information and may result in code motions, both of which may make *lprof* information unreliable.

Different parts of one line of a source file may be executed different numbers of times (e.g., the **for** loop below); the count corresponds to the first part of the line. For example, in the following **for** loop

```
1  [8]   for (j = 0; j < 5; j++)
5  [9]       sub(j);
```

line 8 consists of three parts. The line count listed, however, is for the initialization part, i.e., j = 0.

lprof incorrectly handles the statement immediately following a **for** loop containing a single **if** statement. In the following example, line 8 is executed only once.

```
1  [5]   for (i = 0; i < 3; i++)
3  [6]       if (i > 3)
0  [7]           x = i;
3  [8]   i = 0;
```

This problem can be solved by adding curly braces, as follows:

```
1  [5]   for (i = 0; i < 3; i++) {
3  [6]       if (i > 3)
0  [7]           x = i;
3  [8]   }
1  [9]   i = 0;
```

lprof then handles the statement following the **for** loop correctly.

lprof does not provide execution information about **case** statements containing only a break statement, or about **return** statements without a value.

```
1  [4]   switch (i) {
             case 0:
                 break;
             default:
0  [8]           i = 0;
         }
1  [11]  if (i != 0)
             return;
```

NAME

m4 – macro processor

SYNOPSIS

m4 [options] [files]

DESCRIPTION

The *m4* command is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is –, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

–e Operate interactively. Interrupts are ignored and the output is unbuffered.

–s Enable line sync output for the C preprocessor (#line ...)

–B*int* Change the size of the push-back and argument collection buffers from the default of 4,096.

–H*int* Change the size of the symbol table hash array from the default of 199. The size should be prime.

–S*int* Change the size of the call stack from the default of 100 slots. Macros take three slots, and non-macro arguments take one.

–T*int* Change the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any file names and before any **–D** or **–U** flags:

–D*name*[=*val*]
Defines *name* to *val* or to null in *val*'s absence.

–U*name*
Undefines *name*.

Macro calls have the form:

    name(arg1,arg2, ..., argn)

The **(** must immediately follow the name of the macro. If the name of a defined macro is not followed by a **(**, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

Leading unquoted blanks, tabs, and new-lines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

NAME

mkshlib – create a shared library

SYNOPSIS

mkshlib -s specfil **-t** target [**-h** host] [**-n**] [**-L** dir ...] [**-q**]

DESCRIPTION

mkshlib builds both the host and target shared libraries. A shared library is similar in function to a normal, non-shared library, except that programs that link with a shared library will share the library code during execution, whereas programs that link with a non-shared library will get their own copy of each library routine used.

The host shared library is an archive that is used to link-edit user programs with the shared library [see *ar*(4)]. A host shared library can be treated exactly like a non-shared library and should be included on *cc*(1) command lines in the usual way [see *cc*(1)]. Further, all operations that can be performed on an archive can also be performed on the host shared library.

The target shared library is an executable module that is bound into the user's address space during execution of a program using the shared library. The target shared library contains the code for all the routines in the library and must be fully resolved. The target will be brought into memory during execution of a program using the shared library, and subsequent processes that use the shared library will share the copy of code already in memory. The text of the target is always shared, but each process will get its own copy of the data.

The user interface to *mkshlib* consists of command line options and a shared library specification file. The shared library specification file describes the contents of the shared library.

The *mkshlib* command invokes other tools such as the archiver, *ar*(1), the assembler, *as*(1), and the link editor, *ld*(1). Tools are invoked through the use of *execvp* [see *exec*(2)], which searches directories in the user's PATH. Also, prefixes to *mkshlib* are passed in the same manner as prefixes to the *cc*(1) command, and invoked tools are given the prefix, where appropriate. For example, *i386mkshlib* will invoke *i386ld*.

The following command line options are recognized by *mkshlib*:

-s *specfil* Specifies the shared library specification file, *specfil*. This file contains the information necessary to build a shared library. Its contents include the branch table specifications for the target, the path name in which the target should be installed, the start addresses of text and data for the target, the initialization specifications for the host, and the list of object files to be included in the shared library (see details below).

-t *target* Specifies the output filename of the target shared library being created. It is assumed that this file will be installed on the target machine at the location given in the specification file (see the **#target** directive below). If the **-n** option is used, then a new target shared library will not be generated.

-h *host* Specifies the output filename of the host shared library being created. If this option is not given, then the host shared library will not be produced.

-n Do not generate a new target shared library. This option is useful when producing only a new host shared library. The **-t** option must still be supplied since a version of the target shared library is needed to build the host shared library.

-L *dir ...* Change the algorithm of searching for the host shared libraries specified with the **#objects noload** directive to look in *dir* before looking in the default directories. The **-L** option can be specified multiple times on the command line in which case the directories given with the **-L** options are searched in the order given on the command line before the default directories.

-q Quiet warning messages. This option is useful when warning messages are expected but not desired.

The shared library specification file contains all the information necessary to build both the host and target shared libraries. The contents and format of the specification file are given by the directives listed below.

All directives that can be followed by multi-line specifications are valid until the next directive or the end of the file.

#address *sectname address*

Specifies the start address, *address*, of section *sectname* for the target. This directive typically is used to specify the start addresses of the **.text** and **.data** sections. One **#address** per section name is valid. A **#address** directive must be given exactly once for the **.text** section and once for the **.data** section. See the table in the section "The Building Process" in the "Shared Libraries" chapter of the *UNIX System V Programmer's Guide* for standard addresses.

#target *pathname*

Specifies the absolute path name, *pathname*, at which the target shared library will be installed on the target machine. The operating system uses this pathname to locate the shared library when executing **a.out** files that use this shared library. This directive must be specified exactly once per specification file.

#branch

Specifies the start of the branch table specifications. The lines following this directive are taken to be branch table specification lines.

Branch table specification lines have the following format:

funcname <white space> *position*

where *funcname* is the name of the symbol given a branch table entry and *position* specifies the position of *funcname*'s branch table entry. *position* may be a single integer or a range of integers of the form *position1-position2*. Each *position* must be greater than or equal to one, the same position can not be specified more than once, and every position, from one to the highest given position must be accounted for.

If a symbol is given more than one branch table entry by associating a range of positions with the symbol or by specifying the same symbol on more than one branch table specification line, then the symbol is defined to have the address of the highest associated branch table entry. All other branch table entries for the symbol can be thought of as "empty" slots and can be replaced by new entries in future versions of the shared library. Only functions should be given branch table entries, and those functions must be **external** symbols.

This directive must be specified exactly once per shared library specification file.

#objects

The lines following this directive are taken to be the list of input object files in the order they are to be loaded into the target. The list simply consists of each path name followed by a newline character. This list is also used to determine the input object files for the host shared library, but the order for the host is given by running the list through *lorder*(1) and *tsort*(1).

This directive must be specified exactly once per shared library specification file.

#objects noload

The **#objects noload** is followed by a list of host shared libraries. These libraries are searched in the order listed to resolve undefined symbols from the library being built. During the search it is considered an error if a non-shared version of a symbol is found before a shared version of the symbol.

Each name given is assumed to be a pathname to a host or an argument of the form **-lX**where **libX.a** is the name of a file in LIBDIR or LLIBDIR. This behavior is identical to that of *ld* , and the **-L** option can be used on the command line to specify other directories in which to locate these archives.

Note that if a host shared library is specified using **#objects noload**, any *cc* command that links to the shared library being built will need to specify that host also.

#hide linker [*]

This directive changes symbols that are normally **external** into **static** symbols, local to the library being created. A regular expression may be given [**sh**(1), **find**(1)], in which case all **external** symbols matching the regular expression are hidden; the **#export** directive (see below) can be used to counter this effect for specified symbols.

The optional "*" is equivalent to the directive

```
#hide linker
   *
```

and causes all **external** symbols to be made into **static** symbols.

All symbols specified in **#init** and **#branch** directives are assumed to be **external** symbols, and cannot be changed into **static** symbols using the **#hide** directive.

#export linker [*]

Symbols given in the **#export** directive are **external** symbols (global among files) that, because of a regular expression in a **#hide** directive, would otherwise have been made **static**. For example,

```
#hide linker *
#export linker
   one
   two
```

causes all symbols except *one*, *two*, and those used in **#branch** and **#init** entries to be tagged as **static** .

#init *object*

Specifies that the object file, *object*, requires initialization code. The lines following this directive are taken to be initialization specification lines.

Initialization specification lines have the following format:

ptr <white space> *import*

ptr is a pointer to the associated imported symbol, *import*, and must be defined in the current specified object file, *object*. The initialization code generated for each such line is of the form:

```
ptr = &import;
```

All initializations for a particular object file must be given once and multiple specifications of the same object file are not allowed.

#ident *string*

Specifies a string, *string*, to be included in the .comment section of the target shared library.

##

Specifies a comment. All information on the line beginning with **##** is ignored.

FILES

*TEMPDIR/**	temporary files

TEMPDIR is usually /usr/tmp but can be redefined by setting the environment variable TMPDIR [see *tempnam*() in *tmpnam*(3S)].

NAME

/usr/adm/loginlog – log of failed login attempts

DESCRIPTION

After five unsuccessful login attempts, all the attempts are logged in the *loginlog* file. This file contains one record for each failed attempt. Each record contains the following information:

login name
tty specification
time

This is an ASCII file. Each field within each entry is separated from the next by a colon. Each entry is separated from the next by a new-line.

By default, *loginlog* does not exist, so no logging is done. To enable logging, the log file must be created with read and write permission for owner only. Owner must be **root** and group must be **sys**.

FILES

/usr/adm/loginlog

SEE ALSO

login(1), passwd(1), passwd(1M) in the *User's/System Administrator's Reference Manual.*

NAME

mdevice – file format.

SYNOPSIS

mdevice

DESCRIPTION

The *mdevice* file is included in the directory */etc/conf/cf.d*. It includes a one-line description of each device driver and configurable software module in the system to be built [except for file system types, see *mfsys*(4)]. Each line in *mdevice* represents the *Master* file component from a Driver Software Package (DSP) either delivered with the base system or installed later via *idinstall*.

Each line contains several whitespace-separated fields; they are described below. Each field must be supplied with a value or a '–' (dash).

1. *Device name*: This field is the internal name of the device or module, and may be up to 8 characters long. The first character of the name must be an alphabetic character; the others may be letters, digits, or underscores.

2. *Function list*: This field is a string of characters that identify driver functions that are present. Using one of the characters below requires the driver to have an entry point (function) of the type indicated. If no functions in the following list are supplied, the field should contain a dash.

   o – open routine

   c – close routine

   r – read routine

   w – write routine

   i – ioctl routine

   s – startup routine

   x – exit routine

   f – fork routine

   e – exec routine

   I – init routine

   Note that if the device is a 'block' type device (see field 3. below), a *strategy* routine and a *print* routine are required by default.

3. *Characteristics of driver*: This field contains a set of characters that indicate the characteristics of the driver. If none of the characters below apply, the field should contain a dash. The legal characters for this field are:

   i – The device driver is installable.

   c – The device is a 'character' device.

b – The device is a 'block' device.

t – The device is a tty.

o – This device may have only one *sdevice* entry.

r – This device is required in all configurations of the Kernel. This option is intended for drivers delivered with the base system only. Device nodes (special files in the */dev* directory), once made for this device, are never removed. See *idmknod*.

S – This device driver is a STREAMS module.

H – This device driver controls hardware. This option distinguishes drivers that support hardware from those that are entirely software (pseudo-devices).

G – This device does not use an interrupt though an interrupt is specified in the *sdevice* entry. This is used when you wish to associate a device to a specific device group.

O – This option indicates that the IOA range of this device may overlap that of another device.

4. *Handler prefix*: This field contains the character string prepended to all the externally-known handler routines associated with this driver. The string may be up to 4 characters long.

5. *Block Major number*: This field should be set to zero in a DSP *Master* file. If the device is a 'block' type device, a value will be assigned by *idinstall* during installation.

6. *Character Major number*: This field should be set to zero in a DSP *Master* file. If the device is a 'character' type device (or 'STREAMS' type), a value will be assigned by *idinstall* during installation.

7. *Minimum units*: This field is an integer specifying the minimum number of these devices that can be specified in the *sdevice* file.

8. *Maximum units*: This field specifies the maximum number of these devices that may be specified in the *sdevice* file. It contains an integer.

9. *DMA channel*: This field contains an integer that specifies the DMA channel to be used by this device. If the device does not use DMA, place a '–1' in this field.

SPECIFYING STREAMS DEVICES AND MODULES

STREAMS modules and drivers are treated in a slightly different way from other drivers in all UNIX systems, and their configuration reflects this difference. To specify a STREAMS device driver, its *mdevice* entry should contain both an 'S' and a 'c' in the *characteristics* field (see 3. above). This indicates that it is a STREAMS driver and that it requires an entry in the UNIX kernel's *cdevsw* table, where STREAMS drivers are normally configured into the system.

A STREAMS module that is not a device driver, such as a line discipline module, requires an 'S' in the *characteristics* field of its *mdevice* file entry, but should not include a 'c', as a device driver does.

SEE ALSO

mfsys(4), sdevice(4).

idinstall(1m) in the *User's/System Administrator's Reference Manual.*

NAME

mtune – file format.

SYNOPSIS

mtune

DESCRIPTION

The *mtune* file contains information about all the system tunable parameters. Each tunable parameter is specified by a single line in the file, and each line contains the following whitespace-separated set of fields:

1. *parameter name*: A character string no more than 20 characters long. It is used to construct the preprocessor "#define's" that pass the value to the system when it is built.
2. *default value*: This is the default value of the tunable parameter. If the value is not specified in the *stune* file, this value will be used when the system is built.
3. *minimum value*: This is the minimum allowable value for the tunable parameter. If the parameter is set in the *stune* file, the configuration tools will verify that the new value is equal to or greater than this value.
4. *maximum value*: This is the maximum allowable value for the tunable parameter. If the parameter is set in the *stune* file, the configuration tools will check that the new value is equal to or less than this value.

The file *mtune* normally resides in */etc/conf/cf.d*. However, a user or an add-on package should never directly edit the *mtune* file to change the setting of a system tunable parameter. Instead the *idtune* command should be used to modify or append the tunable parameter to the *stune* file.

In order for the new values to become effective the UNIX system kernel must be rebuilt and the system must then be rebooted.

SEE ALSO

stune(4).

idbuild(1m), idtune(1m) in the *User's/System Administrator's Reference Manual.*

NAME

passwd – password file

DESCRIPTION

passwd contains for each user the following information:

login name
dummy password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
initial working directory
program to use as shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the shell field is null, the default shell is used.

This file has user login information, and has general read permission. It can therefore be used, for example, to map numerical user IDs to names.

The dummy password field consists of the character **x**. This field remains only for compatibility reasons.

FILES

/etc/passwd
/etc/shadow

SEE ALSO

getpwent(3C), group(4).

passwd(1), passwd(1M), login(1) in the *User's/System Administrator's Reference Manual.*

NAME

stune – file format.

SYNOPSIS

stune

DESCRIPTION

The *stune* file contains local system settings for tunable parameters. The parameter settings in this file replace the default values specified in the *mtune* file, if the new values are within the legal range for the parameter specified in *mtune*. The file contains one line for each parameter to be reset. Each line contains two whitespace-separated fields:

1. *parameter name*: This is the name of the tunable parameter used in the *mtune* file.
2. *value*: This field contains the new value for the tunable parameter.

The file *stune* normally resides in */etc/conf/cf.d*. However, a user or an add-on package should never directly edit the *stune* file. Instead, the *idtune* command should be used.

In order for the new values to become effective the UNIX kernel must be rebuilt and the system must then be rebooted.

SEE ALSO

mtune(4).

idbuild(1m), idtune(1m) in the *User's/System Administrator's Reference Manual.*

NAME

syms – common object file symbol table format

SYNOPSIS

#include <syms.h>

DESCRIPTION

Common object files contain information to support symbolic software testing [see *sdb*(1)]. Line number entries, *linenum*(4), and extensive symbolic information permit testing at the C *source* level. Every object file's symbol table is organized as shown below.

```
File name 1.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 1.

File name 2.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 2.
...

Defined global symbols.
Undefined global symbols.
```

The entry for a symbol is a fixed-length structure. The members of the structure hold the name (null padded), its value, and other information. The C structure is given below.

```
#define SYMNMLEN    8
#define FILNMLEN    14
#define DIMNUM      4

struct syment
{
   union                                  /* all ways to get symbol name */
   {
      char        _n_name[SYMNMLEN]; /* symbol name */
      struct
      {
         long     _n_zeroes;      /* == 0L when in string table */
         long     _n_offset;      /* location of name in table */
      } _n_n;
      char        *_n_nptr[2];    /* allows overlaying */
   } _n;
   long           n_value;        /* value of symbol */
```