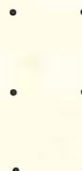
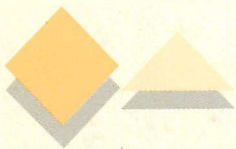


• • •
• •
•

SCO® UNIX® Development System

Developer's Overview



SCO[®] UNIX[®] **Development System**

Developer's Overview



Portions © 1980, 1981-1990 Microsoft Corporation.

All rights reserved.

Portions © 1989, 1990 AT&T.

All rights reserved.

Portions © 1983-1990 The Santa Cruz Operation, Inc.

All rights reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal, Santa Cruz, California, 95062, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

The following legend applies to all contracts and subcontracts governed by the Rights in Technical Data and Computer Software Clause of the United States Department of Defense Federal Acquisition Regulations Supplement:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 52.227-7013. The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, California 95061, U.S.A.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.

DEC, PDP, VAX, and VT100 are trademarks of Digital Equipment Corporation.

Intel is a registered trademark of Intel Corporation.

UNIX is a registered trademark of AT&T.

Date: 10 December 1991

Document Version: 3.2.4D

Preface **1**

Documentation conventions	2
Development System documentation set	3

Chapter 1

Introduction **7**

C language development systems	8
SCO UNIX Development System	8
Source code packages	10

Chapter 2

Using programming tools **11**

Creating the source code	12
Archiving the code	13
Analyzing the code	15
Archiving changes made during the analysis	21
Compiling the code	22
Testing and debugging the code	23
Debugging the code	25
Program structure and guidelines	29

Chapter 3

Common questions **35**

Getting help	35
Installing	36
Preprocessing	39
Compiling	42
Linking	45
Debugging	48
Functions	50
Libraries	55
Manipulating the console and terminals	62
Using curses	65

Creating device drivers	72
Files	74
Cross development	76
Using make	79
Miscellaneous	83

Preface

The *Developer's Overview* provides general and introductory information about the features and facilities available within SCO UNIX System V/386 Development System. This manual is divided into the following chapters:

Chapter 1, "Introduction," provides a general introduction to the products contained in the Development System.

Chapter 2, "Using programming tools," presents some of the tools available within the Development System. A sample program is created, using proper programming techniques, and analyzed with various tools. The rest of the chapter is devoted to debugging the sample program with the **dbxtra** debugger. The debugging session is documented, accompanied with a description of the commands used in the session. Also covered in this chapter are basic program structure and guidelines.

Chapter 3, "Common question" presents commonly asked questions and answers.

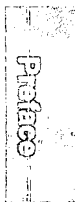
Documentation conventions

SCO documents use font changes and other typographic conventions to distinguish text elements. The following table shows these conventions:

Font conventions

Example	Entity
cc or cc(CP)	command. The “CP” indicates the manual page section in which the command is documented.
<i>libc.a</i>	filename
<i>action</i>	placeholder
⟨Esc⟩	keyname
open or open(S)	system calls, library routines, kernel functions, C keywords. The “S” indicates the manual page section in which the command is documented.
buf	structure
b.errno	structure member
\$HOME	environment or shell variable
SIGHUP	named constant
login	output
“clobber”	jargon
“adm3a”	data value
date	user input

Development System documentation set



The contents of the SCO UNIX System V/386 Development System documentation set are illustrated here. In addition to these books, one set of *Release and Installation Notes* is shipped with each copy of the Development System software.

Overview	<i>Developer's Overview</i>	
Reference	<i>Programmer's Reference Manual (2 volumes)</i>	<i>Encyclopedia</i>
Guides	<i>Programming Tools Guide</i>	<i>Developer's Topics</i>
	<i>Debugging Tools Guide</i>	<i>User Interfaces Guide</i>
Optional	<i>Device Driver Writer's Guide</i>	<i>Macro Assembler Writer's Guide</i>

The books included with the Development System are described here.

Developer's Overview

introduces the Development System facilities and gives general information about developing software to run on SCO UNIX systems and the supported cross-development environments

Programmer's Reference Manual

a two-volume set that includes manual pages for the entire Development System, including sections (CP), (DOS), (FP), (S), and (XNX). See the "Manual pages" article in the *Encyclopedia* for a description of these manual page sections.

Encyclopedia

contains articles that give background information about system internals, descriptions of facilities, and other general issues. Articles are arranged alphabetically.

Developer's Topics

a collection of technical papers about topics of interest to users of the Development System. Many of these papers include extensive examples that illustrate features that SCO added to the porting base.

Programming Tools Guide

provides generally useful information about programming tools and their use in developing software. It also provides implementation-specific details about the ANSI-conforming C compiler that is provided with the Development System.

Debugging Tools Guide

provides generally useful information about the debugging tools, and their use in tracking down and eradicating problems in C and assembly language programs. In addition, it includes information on using **dbXtra** to debug C and C++ programs in a Motif-based windowing environment.

User Interfaces Guide

introduces the facilities that are available for developing user interfaces, and gives detailed instructions for using **curses(S)**, the Extended Terminal Interface (ETI), and writing user interfaces that can be run on ASCII terminals. This book includes numerous examples of **curses** programs.

In addition, the following books are available for Development System customers who need them:

Device Driver Writer's Guide

Provides guide and reference information describing how to write device drivers for SCO UNIX systems. This book is sold separately.

Macro Assembler Writer's Guide

Provides information on the assembly language for the Intel 286, 386, and 486 processors. Developers who are writing device drivers or applications that have tight performance requirements may want to write portions in assembly language. Return the coupon enclosed with the Development System documentation to receive a copy.

Using the Documentation

We recommend the following approach to using this documentation set. In general, we suggest the following:

1. All users should read the *Developer's Overview* thoroughly, soon after installing the software. Experienced users may skim some sections, but this book gives a comprehensive overview of the Development System's capabilities.
2. Use the *Encyclopedia* to look up topics about which you need more information. The *Encyclopedia* gives both summary and background information on many topics, and includes cross-references to sources of more in-depth information.
3. The *Programmer's Reference Manual* contains complete reference information about all capabilities provided in the Development System. New users are warned, however, that the manual pages provide reference material that may be difficult to utilize if they do not understand basic concepts.



The *Programmer's Reference Manual* contains separate sections for Development System commands (CP), system services (S), Development System file formats (FP), and other topics. The pages in each section of the *Programmer's Reference Manual* are arranged alphabetically, although some related facilities are grouped together on a page. If you cannot locate the correct manual page, use the permuted index to find the one that contains the information you want. The "Manual pages" article in the *Encyclopedia* explains how to use the permuted index.

4. The *Programming Tools Guide* and *Debugging Tools Guide* are mainly for users who have not used one of these tools before, or who want to reacquaint themselves with some aspect of that tool's operation. These books are good places to look when the *Encyclopedia* and *Programmer's Reference Manual* are giving you lots of detail, but not enough guidance. Nonetheless, even experienced programmers will discover helpful hints and practical tips on the tools discussed in these books.

Commercial books and articles

A number of fine books and articles are published commercially that discuss how to develop software on the UNIX Operating System. We have not attempted to replicate guide information on all of these topics; manual pages are provided, and many articles in the *Encyclopedia* list additional sources of information on their subject matter. In addition, the *Encyclopedia* includes a large "Bibliography" that will interest users of the Development System.

In particular, we recommend that all developers have the following standard textbooks on their shelves:

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd Edition.

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *UNIX Network Programming*.

Maurice Bach, *Design of the UNIX Operating System*

See the "Bibliography" in the *Encyclopedia* for full citations if you are not familiar with these books.

Chapter 1

Introduction

The Santa Cruz Operation markets four basic classifications of software:

runtime products

a basic environment on which various application software can be executed. Runtime products supplied by SCO include: TCP, NFS, LMC, MERGE, and others.

development products

provide libraries, header files, and tools used to develop user-level applications and kernel-level device drivers

source code products

provide the full source code for development and runtime products

support level supplements SLS

custom-installable disks that contain fixes to SCO products. They can be ordered through the SCO Media Department and Support organizations.

Development System products provide an environment for application development or migration. The following sections discuss the contents of the various Development System products available.

C language development systems

The C language development systems are the core of all development systems. The XENIX and UNIX operating systems are themselves written in C, and one of these operating systems must be installed as a prerequisite for any other development system software.

Three C language development systems are available:

SCO UNIX Development System

provides header files, libraries, tools, and documentation required to develop user-level applications that run on UNIX operating systems, as well as cross-development tools that enable you to create executable files that execute under the XENIX and DOS environments. The UNIX Development System runs on SCO UNIX System V/386 or SCO Open Desktop.

SCO Open Desktop Development System

provides all features of the XENIX and UNIX Development Systems, plus the header files, libraries, tools, and documentation required to write applications that use the X Window and OSF/Motif toolkits, TCP/IP, and Embedded SQL facilities

SCO XENIX Development System

provides facilities similar to those in the UNIX Development System, but runs on the XENIX Operating System or SCO Open Desktop.

SCO UNIX Development System

The Development System consists of the following components:

PERM	permlists
SOFT	basic software development tools
LEX	generates programs for lexical analysis
YACC	Yet Another Compiler-Compiler
CFLOW	generates C flow graphs
LINT	syntax and usage check files and tools
SCCS	Source Code Control System
DOSDEV	DOS cross-development utilities and libraries
OS2DEV	OS/2 cross-development libraries and utilities

XNXDEV	XENIX cross-development include files and utilities
386XDEV	XENIX 386 cross-development libraries
286XDEV	XENIX 286 cross-development libraries
MESGCAT	XPG Message Catalog utilities
MAN	manual pages
XTI	XTI Development System
DBXTRA	dbXtra distribution
ISAM	SCO ISAM libraries
MMDF	Multi-channel Memorandum Distribution Facility



Changes and additions

Under the C language section, support for long filenames has been included for this release. All Development System libraries and utilities that access data files (COFF, AR, OMF) have been updated to support long filenames. For a complete list of the affected system utilities, see the *Release and Installation Notes*.

MMDF provides a well-defined Application Program Interface (API) for both mail reading programs and mail channels (like the smtp client). The functions in *libmmdf.a* allow developers to create third party channels to support new mail transfer protocols and mail readers that take advantage of the address and syntax checking provided by the base MMDF system. *libmmdf.a* also allows developers to access existing third-party applications that already understand the MMDF mail system (such as the MH mail user agent).

A new release of the C compiler is included. Within this compiler release, the libraries and header files have been modified to provide full conformance with the following standards:

- X/Open Portability Guide, Issue 3, 1989
- ANSI X3.159-1989 Programming Language — C
- IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1)
- AT&T SVID Issue 2
- Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2)
- NIST FIPS 151-1

Source code packages

The Santa Cruz Operation offers source code to the following products:

- XENIX and UNIX Runtime and Development Systems
- SCO Open Desktop Development System

Chapter 2

Using programming tools

The Development System includes a powerful set of programming tools to compile, analyze, and debug programs. This chapter describes how to use some of these tools to develop a small sample program. By the end of the chapter, you will have a working knowledge of `SCCS`, `cb`, `cscope`, `lint`, `cflow`, `cxref`, `cc`, `make`, and `dbXtra`. Check the manual pages in the CP section for additional tools that may be of use in other situations. Each manual page lists the supported options for each command.

For the purposes of this example, program development proceeds as follows:

- create the source code
- archive a copy of the original code
- analyze the code
- compile the code
- test and debug the code
- archive any changes made during testing and debugging

The Development System provides tools to help at each stage of the development process. This chapter describes their use and also provides general information and guidelines for structuring programs and applications.

Creating the source code

Program development begins with the creation of code from specifications. Sometimes, the code has already been written and must only be tested and debugged. This tutorial includes a sample program.

The sample program, *testcase.c*, was written to these specifications:

- Read input from `stdin` and write output to `stdout`.
- For each line of input, output the words that comprise the line, one word to a line. A word is defined to be any sequence of alphabetic characters separated by a white space character.
- If the optional argument, `-c`, where `c` is a character, is specified, output only those words containing the character.

The file *testcase.c* contains the following code:

```

1      #include <stdio.h>
2      #include <ctype.h>
3      #include <string.h>
4
5      /* Manifests for state machine to parse input line. */
6      #define WORD 0
7      #define IGNORE 1
8
9      /* Globals, used by both subroutines. */
10     char Words[BUFSIZ/2]; /* Worst case, single letters. */
11     int WordCount;
12
13     /* Walk through the array of words, find those with the
14      * matching character, printing them on stdout. Note that
15      * the NULL character will match all words. */
16     void PrintWords(wc, match)
17     int wc; /* Number of words in Words[] */
18     char match; /* Attempt to match this character. */
19     {
20         register int ix; /* Index in Words[] */
21         register char *cp; /* Pointer for searching. */
22         for(ix = 0; ix < wc; ix++) {
23             cp = Words[ix];
24             /* Try to match the given character.
25              * Scan the word, attempting to match,
26              * or until the end of the word is found. */
27             while((*cp) && (*cp++ != match));
28             if (*cp == match) /* Found a match? Write the word on stdout. */
29                 (void) printf("%s0", Words[ix]); } return; }
30
31     /* Find words in the given buffer. The Words[] array is set
32      * to point at words in the buffer, and the buffer modified
33      * with NULL characters to delimit the words. */
34     int GetWords(buf)
35     char buf[]; /* The input buffer. */
36     {
37         register char *cp; /* Pointer for scanning. */
38         int end = strlen(buf); /* Length of the buffer. */
39         register int wc = 0; /* Number of words found. */

```

```

38         int         state = IGNORE;           /* Current state. */
39         /* For each character in the buffer. */
40         for(cp = &buf[0]; cp < &buf[end]; cp++) {
41             /* A simple state machine to process
42              * the current character in the buffer.
43              */
44             switch(state) {
45                 case IGNORE:
46                     if (!isspace(*cp)) {
47                         Words[wc++] = cp; /* Just started a word? Save it. */
48                         state = WORD; /* Reset the state. */ } break;
49
50                 case WORD:
51                     if (isspace(*cp)) {
52                         *cp = '\0'; /* Just completed a word? terminate it. */
53                         state = IGNORE; /* Reset the state. */ } break; } }
54         return wc; /* Return the word count. */ }
55
56 int main(argc, argv) int argc; char *argv[]; { char   buf[BUFSIZ], match;
57 /* Check command line arguments. */
58 if (argc < 2) match = ' ';
59 /* No command line argument, match all words. */
60 else match = *++argv[1]; /* Match the char after the first - */
61 /* Until no more input on stdin. */
62 while(gets(buf) != (char *)NULL) {
63     WordCount = GetWords(buf); /* Parse the input buffer. */
64     PrintWords(WordCount, match); /* Print the matching words. */ }
65 return(0); /* Return success to the shell. */ }

```

Archiving the code

During a software development project, it is often desirable to track the changes made to the source code, who made them, and the reasons for the change. In addition, if more than one person is working on the project, it is important that two or more people don't try to make changes to the code at the same time.

SCCS (Source Code Control System) is a maintenance and enhancement tracking tool that provides these capabilities. When changes are made, SCCS identifies and stores them in the file with the original source code or documentation. Any previous version of the file can be reconstructed for inspection or for additional modification. History data can be stored with each version to help remember why, when, and by whom changes were made.

Storing source files under SCCS also prevents two people from accidentally modifying the same file at the same time. To change a file, it must explicitly be checked out for editing. If a version of the file is already checked out for editing, other developers are prevented from checking it out.

If you want a copy of the original version of *testcase.c* at a later date, it is a good idea to place it under SCCS. This will let you track the changes made to *testcase.c* as it proceeds through the development procedure. To check *testcase.c* into SCCS, issue the following command:

```
admin -itestcase -r1 s.testcase
```

The following message is displayed:

```
No id keywords (cm7)
```

NOTE **admin** issues a warning message: No id keywords (cm7). Study the SCCS documentation for information about how to supply an ID keyword.

SCCS creates a file with the name specified on the command line, in this case *s.testcase*. SCCS adds some lines to the beginning of the file:

```
^Ah22947
^As 00117/00000/00000
^Ad D 1.1 91/10/07 13:26:29 username 1 0
^Ac date and time created 91/10/07 13:26:29 by username
^Ae
^Au
^AU
^At
^AT
^AI 1

#include<stdio.h>
#include<ctype.h>
```

NOTE In the previous output, the **^A** denotes a **(Ctrl)A**.

Once *s.testcase* is under SCCS control, the file must be checked out for editing before any changes can be made. To make changes to *s.testcase*, check it out for editing with the following command:

```
get -e s.testcase
```

SCCS responds with the following message:

```
1.1
new delta 1.2
117 lines
```

The result is a file called *testcase*, which is the file we will edit, and a file called *p.testcase*, which SCCS uses to control file access. If anyone tries to access the file once it has been checked out for editing, SCCS issues an error message indicating that the file is already checked out and by whom.

Analyzing the code

If you are working on source code that you didn't write, the purpose of various constructs may not be clear. The Development System provides a number of tools to help make sense out of the source code. In this section, we will use the following tools to analyze the *testcase.c* source code:

- cb** a tool to format C source code
- cscope** a tool to examine functions contained with a program
- lint** a tool to identify code that may not pass through the compiler without an error message or that may be coded using questionable coding practices
- cflow** a tool to identify the nesting levels of functions within a program
- cxref** a tool to create a cross reference of functions and symbols used within a program

Using cb

When developing software, it is easy to forget to format the code so it is both easy to read and the program logic is apparent. Running a program through **cb** (C program beautifier) corrects this problem. For example, some of the code in *testcase* is less than beautiful:

```
55     int main(argc, argv) int argc; char *argv[]; { char  buf[BUFSIZ], match;
56     /* Check command line arguments. */
57     if (argc < 2) match = ' ';
58     /* No command line argument, match all words. */
59     else  match = *++argv[1];    /* Match the char after the first - */
```

While this code fragment uses legal C syntax that is readily understood by the compiler, it is hard for a human to understand. When the code is run through **cb** with the **-s** option specified, the reformatted code is easier to follow and understand:

```
55     int      main(argc, argv)
56     int      argc;
57     char     *argv[];
58     {
59         char   buf[BUFSIZ], match;
60         /* Check command line arguments. */
61         if (argc < 2)
62             match = ' ';
63             /* No command line argument, match all words. */
64         else
65             match = *++argv[1]; /* Match the char after the first - */
```

For more information, see the **cb**(CP) manual page.

Using cscope

The purpose of **cscope** is to build a database containing information about functions, function calls, macros, and variables. **cscope** allows you to look at functions and examine the lines of code around those functions. In the *testcase.c* example, all the functions are contained in the one file. If, however, functions that are called from *testcase.c* were located in other files (excluding system library functions). **cscope** would allow you to look directly at those functions without exiting **cscope**. While using **cscope** on *testcase.c* might not be extremely helpful, it may still be of interest.

To examine the functions within *testcase.c*, run **cscope** as follows:

```
cscope testcase.c
```

To move the cursor to a menu option, press the **<Return>** key. To examine the function **main**, move the cursor to "List functions called by this function:", enter **main**, and press **<Return>**:

```
cscope                                     Press the ? key for help
```

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

The display lists the three functions called by **main**: **gets**, **GetWords** and **PrintWords**. Displayed along with the function names is the source code line that contains the function.

```
Functions called by this function: main
```

```
File      Function  Line
1 testcase.c gets      61 while(gets(buf) != (char *)NULL) {
2 testcase.c GetWords   62 WordCount = GetWords(buf);
3 testcase.c PrintWords 63 PrintWords(WordCount, match); }
```

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

By examining the **GetWords** function, we can move deeper into the code:

Functions called by this function: **GetWords**

```
File      Function Line
1 testcase.c strlen  36 int end = strlen(buf);
2 testcase.c isspace 46 if (!isspace(*cp)) {
3 testcase.c isspace 50 if (isspace(*cp)) {
```

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:

List file names containing this text string:

In analyzing the results, we see that **GetWords** calls three functions displayed with their line numbers in the source code.

Using **cscope** is particularly helpful when examining a large or complex program that has many files that span several directories. For more information, see the “C Programmer’s Productivity Tools” chapter in the *Programming Tools Guide* and the **cscope**(CP) manual page.

Using lint

The **lint**(CP) program examines source code to identify sections of code that the compiler may flag as incorrect. In many cases, **lint** identifies as incorrect sections of code that the compiler accepts but that uses poor programming practice that may affect portability and robustness.

To find out how portable and robust our sample program is, run *testcase.c* through **lint** with the command:

```
lint -n testcase.c
```

lint produces the following display:

```
testcase.c(16) : warning C4131: 'PrintWords' : uses old-style declarator
testcase.c(33) : warning C4131: 'GetWords' : uses old-style declarator
testcase.c(55) : warning C4131: 'main' : uses old-style declarator
testcase.c(61) : warning C4601: nonportable character comparison
symbol _cuc1c declared but never used
```

The warning messages indicate that some functions in *testcase* are declared in the old, pre-ANSI style. While this style of function declaration is still acceptable, it is likely to be disallowed in the future. Code using the old style of function declarations is not portable to future C compilers.

lint(CP) prints the warning non-portable character comparison because **NULL** is a special value. It should not be assumed that the value of **NULL** is 0. On some machines, a pointer of value 0 is an illegal address, or is unavailable to regular users. On ANSI C standard systems, **NULL** is defined in *stddef.h* to an "implementation-defined null pointer constant".

The last warning, symbol **_cuclic** declared but never used is the result of the symbol **_cuclic** being declared in */usr/include/ctype.h* but never being used by *testcase.c*. We also never use the macro defined in *ctype.h* in *testcase.c*, which references that symbol.

For more information on using **lint**, see the chapter "Lint" in the *Programming Tools Guide* and the **lint(CP)** manual page.

Using cflow

The **cflow** command presents a list of the functions and function calls within a program. This list can be helpful when debugging a program, especially if there are external references. The information displayed by **cflow** indicates the source filename and the line number of the function, and lists additional functions that are in turn called by that function. To see a list of the functions called within *testcase.c*, we can run **cflow**:

cflow testcase.c

```
1      main: int () <testcase.c 55>
2          gets: <>
3          GetWords: int () <testcase.c 34>
4              strlen: <>
5          PrintWords: void () <testcase.c 17>
6              printf: <>
```

Output is displayed in the following fields:

line number	the number of the line in the output
tabs	the level at which the symbol is defined
symbol name	the name of the symbol under consideration followed by a colon
symbol information	the type of the symbol and the arguments to the symbol, if any
location	the name of the file and the number of the line within the file in which the symbol is defined

In analyzing the output, we see that the function **main** is the first function called by *testcase.c* and is defined on line 55 as shown in the angle brackets (<>). Notice the function **main** returns a type **int** and it calls the functions **gets**, **GetWords**, and **PrintWords**. The <> next to the **gets** function are empty because **cflow** only looks in the source file to resolve function call references.

The `gets` function is not in the source code (*testcase.c*) and, therefore, `cflow` does not have any information on it. The function `main` calls two additional functions, `GetWords` and `PrintWords`. Both of these functions are found in the source file and, therefore, their locations are displayed. The two additional functions, `strlen` and `printf`, called respectively from `GetWords` and `PrintWords`, do not exist within *testcase.c*; hence, no line number information is available for them.

As a result of analyzing *testcase.c* with `cflow`, we now know the names of all of the functions referenced in the source file and the location of their definitions in the source file. If we need to change any of them, they should be easy to find. For a larger more complex program, you can make decisions on where best to optimize code, either by reducing the number of function calls or by identifying obsolete functions and removing them. For more information on using `cflow`, see the `cflow(CP)` manual page.

Using *cxref*

The `cxref` tool produces a list of all functions and symbols (`auto`, `static` and `global`) that can be used as a cross-reference table for large or complex programs. `cxref` is more informative than `cflow` because not only are the function calls that exist within the source file displayed, with line numbers, but all external function calls also have that information. To see a list of the function calls and symbol references used by *testcase.c*, run `cxref`:

```
cxref testcase.c
```



The following output is displayed:

testcase.c

SYMBOL	FILE	FUNCTION	LINE
BUFSIZ	/usr/include/stdio.h--	*91	
	testcase.c	--	10
	testcase.c	main	55
GetWords()	testcase.c	GetWords	*34
	testcase.c	main	62
IGNORE	testcase.c	--	*7
	testcase.c	GetWords	38 45 52
NULL	/usr/include/stdio.h--	*57	
	testcase.c	main	61
PrintWords()	testcase.c	PrintWords	*17
	testcase.c	main	63
WORD	testcase.c	--	*6
	testcase.c	GetWords	48 49
WordCount	testcase.c	--	*11
	testcase.c	main	62 63
Words	testcase.c	--	*10
	testcase.c	PrintWords	22 28
	testcase.c	GetWords	47
_S	/usr/include/ctype.h--	*38	
	testcase.c	GetWords	46 50
__ctype	/usr/include/ctype.h--	*91 97	
	testcase.c	GetWords	46 50
_ctype	/usr/include/ctype.h--	97	
	testcase.c	GetWords	46 50
argc	testcase.c	main	*55 57
argv	testcase.c	main	*55 59
buf	testcase.c	GetWords	*35 36 40
	testcase.c	main	*55 61 62
cp	testcase.c	PrintWords	*20 22 26 27
	testcase.c	GetWords	*35 40 46 47 50
end	testcase.c	GetWords	*36 40
gets	/usr/include/stdio.h--	257	
	testcase.c	main	61
isspace	/usr/include/ctype.h--	72 *113	
	testcase.c	GetWords	46 50
ix	testcase.c	PrintWords	*19 21 22 28
main()	testcase.c	main	*55
	testcase.c	PrintWords	*19 26 27
match	testcase.c	main	*55 57 63
	testcase.c	PrintWords	28
printf	/usr/include/stdio.h--	238	
state	testcase.c	PrintWords	28
	testcase.c	GetWords	*38 44 48 52
strlen	/usr/include/string.h--	97	
wc	testcase.c	GetWords	36
	testcase.c	PrintWords	*19 21
	testcase.c	GetWords	*37 47 53

The first column lists the name of the symbol. The second column lists the name of the file(s) in which the symbol appears. The third column lists the name of the function in which the symbol appears, and the fourth column provides the line number within the file in which the symbol appears.

To illustrate, let's examine the symbol **BUFSIZ** in the **cxref** output. The first entry names the symbol, **BUFSIZ**, and the file where it is declared, indicated by an asterisk (*) next to the line number. The second entry indicates its first use in *testcase* denoted by only a line number. The third and final entry for **BUFSIZ** indicates its use in the function **main** of *testcase.c* on line 55.

The listing produced by **cxref** is helpful when debugging a large complex program. Using the **cxref** output, external function calls can be tracked down with ease. For more information, see **cxref(CP)** manual page.

Archiving changes made during the analysis

During our analysis of the code for *testcase.c*, we modified the sample program; specifically, we beautified the code using **cb**. Because it is always a good idea to track changes to a project, we should check the modified code back into SCCS.

To check the modified code back into SCCS, issue the command:

```
delta s.testcase
```

Supply a brief comment describing the changes in response to the `comment?` prompt.

After the comment has been added, SCCS responds with a brief message indicating what has happened to the file:

```
No id keywords (cm7)
1.2
5 inserted
9 deleted
117 unchanged
```

To list the history of changes to the *testcase* file, issue the following command:

```
prs s.testcase
```

You see:

```
D 1.2 91/10/07 14:32:51 username 2 1      00000/00000/00117
MRs:
COMMENTS:
sample editing changes made to this file

D 1.1 91/10/07 13:26:29 username 1 0      00117/00000/00000
MRs:
COMMENTS:
date and time created 91/10/07 13:26:29 by username
```

There are other tools available to retrieve an earlier version of the file, compare two versions of the file, and so forth. The "Source Code Control System

(SCCS)” chapter in the *Programming Tools Guide* includes a detailed explanation and tutorial for using SCCS.

Compiling the code

Because we have already performed an analysis of the program, the program should compile without errors. Compile *testcase* with the following command line:

```
cc -o testcase testcase.c
```

As expected, the compiler compiles the program without errors and names the resulting executable file *testcase*.

The compiler supports a large number of options that can be used to control how the program is compiled and linked. One that is commonly used is the *-l* option, which specifies the libraries to link in. The default library is the standard C library; if your program uses any routines from other libraries, you will need to specify those libraries on the compile line. The section (S) manual pages indicate which library is required for which routine.

For more information, see the *Programming Tools Guide* and the *cc(CP)* manual page.

Using make

While the procedure for compiling our sample program is relatively simple, the procedure for compiling larger projects can be considerably more complex. Fortunately, the Development System provides a tool to automate the procedure. There are a number of reasons for using the *make(CP)* utility to automate compilation. The manual page the *cc(CP)* describes a large number of options for the compilers. In some instances, the appropriate assortment of libraries, indicated with the *-l* option, needs to be selected each time a program is compiled. Remembering all the options can become tedious; that is one reason for creating a makefile to use when compiling and linking the program. Another advantage of a makefile is that when several files are linked together into one executable file, *make(CP)* only recompiles those files that have been modified (or “touched”) since the last link. This reduces the time required to recompile code.

We can use the following makefile to automate the compilation of *testcase.c*:

```
#
# Makefile for testcase.c
#

# Compile testcase.c normally

testcase: testcase.c
    cc -o testcase testcase.c

# Compile testcase.c to be using in debugging

debug: testdb testerr

testdb: testcase.c
    cc -otestcase -Od -g testcase.c

testdb:
    cc -DERROR -otesterr -Od -g testcase.c
```

The above makefile provides two ways of compiling *testcase.c*. Invoking the makefile with the following command line compiles the file with no debugging information contained in the binary:

make testcase

However, if **make** is invoked with the command line below, debugging information is included.

make testdb

For more information, see the “Make” chapter in the *Programming Tools Guide* and the **make**(CP) manual page.

Testing and debugging the code

Even though *testcase* has compiled without errors, we cannot be sure that the program logic is correct. The only way to ensure that the logic is correct and the program does what is intended, is to test it thoroughly. Any errors revealed by testing must, of course, be corrected.

Testing

Because *testcase* is a small, simple program, we can test it by simply running it on selected input and verifying that the output is what we expect. Larger, more complex programs benefit from a more rigorous testing procedure.

Test *testcase* with the following command line and input (bold typeface indicates user input):

```
testcase  
this is a test of the testcase program  
this  
is  
a  
test  
of  
the  
testcase  
program
```

With no character argument specified, *testcase* responds with one word per line, as expected. Test the option to *testcase* with the following command line and input:

```
testcase -t  
this is a test of the testcase program
```

In this example, with the *-t* as an argument, *testcase* should have printed out only words with the character *t* in it; it did not. This deserves further exploration. Run *testcase* with the following input:

```
testcase -t  
this is a ttest of the testcase program  
ttest
```

testcase seems to be printing out words that have double occurrences of the specified character, *t*. Try it with the following input just to make sure:

```
testcase -t  
attention, the ttime is at hand  
attention,  
ttime
```

That seems to be the problem with our program. Now that we have identified the problem, we have to find out where in the source code the problem occurs. Fortunately, the Development System provides the tools to debug source code.

Debugging the code

SCO UNIX System V supports four debuggers that can be used to help locate errors in source code. Each debugger has its special uses, but in most cases choosing between them is largely a matter of taste.

Using *dbxtra*

To debug *testcase*, use **dbxtra**. For *testcase* to be debugged, it must be compiled with the **cc -g** option specified. This option specifies that the compiler should include symbolic information in the binary. Compile *testcase.c* with debugging information using **make**:

make testdb

To initiate **dbxtra** issue the following command:

dbxtra testcase

The following **dbxtra** screen is displayed.

```

54
-> 55      int main(argc, argv) int argc; char *argv[]; { char      buf[BUFSI
21. match:
      56      /* Check command line arguments. */
57      if (argc < 2) match = '\0';
58      /* No command line argument, match all words. */
59 else  match = *++argv[1]; /* Match the char after the first - */
60      /* Until no more input on stdin. */
61      while(gets(buf) != (char *)NULL) {
62          WordCount = GetWords(buf); /* Parse the input buffe
r. */      63          PrintWords(WordCount, match); /* Print the matchin
C:\browsed\11\testcase.c ... lunc:
      dbXtra version 3.0
      (C) Copyright 1989, 1990 SCO Canada, Inc.
      All rights reserved.
      reading symbolic information ...
      Type 'help' for help.
(dbxtra) █

```

To begin debugging *testcase*, it must be run under **dbxtra**'s control. To execute *testcase* within **dbxtra**, issue the **dbxtra run** command:

```

(dbxtra) run -%
This is a test
So is% this
%%
%%
<Ctrl>d
execution completed
(dbxtra)

```

In this example, we had *testcase* look for the argument **%**, which it failed to match in the second line, but did in the third.

The **cp** pointer points to the characters in the input buffer. Therefore, the **cp** pointer might be a place to start searching for the problem. Because we know the **cp** pointer is used in the **PrintWords** function from the **cxref** output, examining the function **PrintWords**, and monitoring **cp**, might lead us to discover why *testcase* is not behaving correctly. At this point, a breakpoint should be set in the **PrintWords** function. This allows control over the program's execution, enabling us to monitor the **cp** pointer. The following commands set a breakpoint and check to ensure that breakpoint has been set:

```
(dbxtra) func PrintWords
(dbxtra) stop at 22
[1] stop at 22
(dbxtra) status
[1] stop at 22
(dbxtra)

21         for(ix = 0; ix < wc; ix++) {
S 22         cp = Words[ix];
23         /* Try to match the given character.
24         * Scan the word, attempting to match.
25         * or until the end of the word is found. */
26         while((*cp) && (*cp++ != match));
27         if (*cp == match) /* Found a match? Write the word
d on stdout. */
60         28         (void) printf("%s\n", Words[ix]); } r
eturn 29
30         /* Find words in the given buffer. The Words[] array is set
r. */31 /* to point at words in the buffer, and the buffer modified
Browse (C) /testcase.c
dbXtra version 3.0
(C) Copyright 1989, 1990 SCO Canada, Inc.
All rights reserved.
reading symbolic information ...
Type 'help' for help.
(dbxtra) func PrintWords
(dbxtra) stop at 22
[1] stop at 22
(dbxtra) status
[1] stop at 22
(dbxtra) █
```

Notice the **S** beside line 22; this indicates that a breakpoint has been set.

The next action is to rerun the program with the breakpoint in effect:

```
(dbxtra) rerun
Running: testcase -%
% a b c
[1] stopped in PrintWords at line 22
(dbxtra)
```

The **where** command displays the current location in the stack frame:

```
(dbxtra) where
PrintWords(match = '%', wc = 4), line 22 in "testcase.c"
main(0x2, 0x7ffff784, 0x7ffff790), line 65 in "testcase.c"
(dbxtra)
```

The following two **dbxtra** commands display the type of variable (character) and what the match character is (%):

```
(dbxtra) whatis match
char match;
(dbxtra) print match
"% "
(dbxtra)
```

Because the breakpoint was set on line 22, the pointer **cp** has not been set yet. To execute the next step and set **cp**, enter **next** on the **dbxtra** command line. If the pointer **cp** is checked now:

```
(dbxtra) next
(dbxtra) print cp
"% "
```

It displays the correct character. If one more line of code is executed, the **cp** pointer should display the same character:

```
(dbxtra) next
(dbxtra) print cp
" "
(dbxtra)
```

Notice that there is no value currently in **cp**. This may be the problem; it would seem that the **cp** pointer is being incremented even if a match is found. The result is that only double occurrences of the match variable character produces the correct behavior.

The next step in debugging *testcase* is to check if the problem assessment is correct. *testcase* needs to be rerun, and the **cp** pointer set:

```
(dbxtra) rerun
Running: testcase -%
% a b c
[1] stopped in PrintWords at line 22
(dbxtra) next 2
(dbxtra) print cp
" "
(dbxtra)
```

Notice the **cp** pointer is indicating a null character. The next step is to reset the **cp** pointer.

The **assign** command allows variable values to be changed. In this example, by subtracting 1 from **cp**, the correct character variable should be displayed.

```
(dbxtra) assign cp = cp - 1
(dbxtra) next
(dbxtra) next
%
(dbxtra)
```

This indicates that the problem assessment was correct. The pointer was being incremented regardless of whether or not a match condition existed.

The last step is to delete the breakpoint and complete the program's execution:

```
(dbxtra) cont
[1] stopped in PrintWords at line 22
(dbxtra) delete 1
(dbxtra) cont
<Ctrl>d
execution completed
(dbxtra) q
```

Correcting the code

To fix the problem within the *testcase.c* code, *testcase.c* must be checked out of SCCS, changed, recompiled, and retested. This ensures that the logic problem located through the debugger is the only problem with the code.

Check *s.testcase* out of SCCS:

```
get -e s.testcase
```

SCCS responds with the following message:

```
1.2
new delta 1.3
117 lines
```

Edit *testcase.c* by locating the following line in the *PrintWords* function:

```
while((*cp) && (*cp++ != match))
if (*cp == match) /* Found a match? Write the word on stdout. */
    (void) printf("%s0, Words[ix]); } return; }
```

Change the line to read as follows:

```
while((*cp) && (*cp != match))
    ++cp;
if (*cp == match) /*Found a match? Write the word on stdout. */
    (void) printf("%s0, Words[ix]); } return; }
```

Save the file, exit the editor, and check *testcase* back into SCCS:

```
delta s.testcase
```

The new version of *testcase.c* now needs to be recompiled. Execute the makefile to ensure all compiling and linking steps are executed:

```
make testcase
```

Testing the correction

The program *testcase* should be retested to ensure the corrected pointer causes the correct behavior.

```
testcase
this is a test of the testcase program
this
is
a
test
of
the
testcase
program
```

With no character argument specified, *testcase* responds with one word per line. This is the correct behavior.

```
testcase -t
this is a test of the testcase program
this
test
testcase
```

In this example, with the *-t* as an argument, *testcase* has printed out, one word per line, only words with the letter *t*. This is the correct behavior. So the assumption that the pointer was incrementing regardless of whether or not a match condition existed was correct. Changing it solved the problem within *testcase*.

Program structure and guidelines

As illustrated through *testcase*, specification of applications must define the actual work to be accomplished. There are many issues that should be considered in designing applications written in C, including:

- modularity Good C programming style suggests that large applications be coded in several small component parts. Several C codes can be linked together to form one executable file; use the **make(CP)** utility to simplify the process.
- functions Functions that could run parallel to each other should be coded in separate executable files. This is especially useful if you ever port the application to a multiprocessor distributed system.
- archives If your application uses certain routines frequently in different programs, it is advisable to create an archive library or shared library for those routines. Shared libraries usually reduce the amount of disk space and memory occupied by the executable

modules and enable you to update all programs that call a certain routine without recompiling those programs. There are limits to the number of shared libraries that any executable program can access. For more information, see the “Shared libraries” chapter in the *Programming Tools Guide*.

Program structure

The descriptions that follow use the line numbers within *testcase* to indicate their location. This is an example of correct programming structure.

```

1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4
5 /* Manifests for state machine to parse input line. */
6 #define WORD 0
7 #define IGNORE 1
8
9 /* Globals, used by both subroutines. */
10 char *Words[BUFSIZ/2]; /* Worst case, single letters. */
11 int WordCount;
12
13 /* Walk through the array of words, find those with the
14  * matching character, printing them on stdout. Note that
15  * the NULL character will match all words. */
16 void PrintWords(wc, match)
17 int wc; /* Number of words in Words[] */
18 char match; /* Attempt to match this character. */
19 {
20     register int ix; /* Index in Words[] */
21     register char *cp; /* Pointer for searching. */
22     for (ix = 0; ix < wc; ix++) {
23         cp = Words[ix];
24         /* Try to match the given character.
25          * Scan the word, attempting to match,
26          * or until the end of the word is found. */
27         while ((*cp) && (*cp++ != match))
28             ;
29         if (*cp == match) /* Found a match? Write the word on stdout. */
30             (void) printf("%s0", Words[ix]);
31     }
32     return;
33 }
34
35
36
37 /* Find words in the given buffer. The Words[] array is set
38  * to point at words in the buffer, and the buffer modified
39  * with NULL characters to delimit the words. */
40 int GetWords(buf)
41 char buf[]; /* The input buffer. */
42 {
43     register char *cp; /* Pointer for scanning. */
44     int end = strlen(buf); /* Length of the buffer. */
45     register int wc = 0; /* Number of words found. */

```

```

46         int         state = IGNORE;           /* Current state. */
47         /* For each character in the buffer. */
48         for (cp = &buf[0]; cp < &buf[end]; cp++) {
49             /* A simple state machine to process
50              * the current character in the buffer.
51              */
52             switch (state) {
53                 case IGNORE:
54                     if (!isspace(*cp)) {
55                         Words[wc++] = cp; /* Just started a word? Save it. */
56                         state = WORD; /* Reset the state. */
57                     }
58                     break;
59                 case WORD:
60                     if (isspace(*cp)) {
61                         *cp = '\0'; /* Just completed a word? terminate it. */
62                         state = IGNORE; /* Reset the state. */
63                     }
64                     break;
65             }
66         }
67         return wc; /* Return the word count. */
68     }
69
70
71
72 int     main(argc, argv)
73 int     argc;
74 char    *argv[];
75 {
76     char    buf[BUFSIZ], match;
77     /* Check command line arguments. */
78     if (argc < 2)
79         match = ' ';
80         /* No command line argument, match all words. */
81     else
82         match = *++argv[1]; /* Match the char after the first - */
83     /* Until no more input on stdin. */
84     while (gets(buf) != (char *)NULL) {
85         WordCount = GetWords(buf); /* Parse the input buffer. */
86         PrintWords(WordCount, match); /* Print the matching words. */
87     }
88     return(0); /* Return success to the shell. */
89 }

```

Lines 1-3 Begin the code with **#include** statements for the header files that are required. Each section (S) page lists the header files that are required for that particular call; use a **#include** line for each one that is needed by the calls you are using.

There are some rules about the order in which header files are included; see the “Header files” section later in this chapter and the articles about each header file in the *Encyclopedia*. Application-specific header files should always be included after the standard header files.

Lines 55-64

The `main()` routine controls the order in which the routines execute. It is best to keep `main()` fairly succinct. Keep complex functionality in subroutines, located either before or after `main()`. This makes it easier to look at `main()` to get the general notion of program flow. If the application has several programs that call some of the same routines, those shared routines can be located in an archive or shared library for retrieval in all programs.

Line 55

The `argc` and `argv` parameters to `main()` capture arguments given on the command line. When a program begins execution, `argc` contains the argument count and `argv` is an array of pointers to the argument values. An argument is stored as a null-terminated string. The argument count is never less than 1, because the program name is always considered the first argument. See the "Argument" article in the *Encyclopedia* for more information.

Lines 16-53

Subroutines are coded outside the `main()` routine. Some people prefer to put them before `main()`, and others prefer to put them afterwards. Either approach works equally well.

Each subroutine should have a block comment preceding it to describe the subroutine and provide information about any non-obvious coding that may have been used. Well-written comments will significantly reduce the effort required to maintain the code in the future. Good C programming style dictates that block comments are not spread over several lines to the right of the actual code. Comments can be put on the line with the code, but these should be short comments that are specific to that line of code, such as shown in lines 17 and 18.

Header files

Header files contain information such as the following:

- global **defines** that may need to be changed or that are shared by several programs and/or drivers
- structure definitions
- macro definitions
- hardware-dependent information that may need to be modified when porting to another system

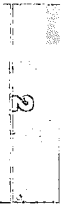
Header files are called on `#include` lines at the beginning of the code. They must conform to C programming style, since they are accessed as part of the code by the compiler. In general, `#include` statements are placed at the beginning of the code to ensure that they affect the entire code. However, if the intended scope of a particular `#include` line is only specific to a few lines of code, that `#include` line can be located anywhere throughout the code.

The Development System contains many header files under the `/usr/include` directory. Header files accessed only by user-level code are not in the `/usr/include` directory itself, and header files accessed by drivers and other kernel-level code are in the `/usr/include/sys` directory. All header files created specifically for an application should be stored in that source tree so they can be retrieved easily after a software upgrade replaces the contents of the `/usr/include` tree. This is important if the product is to be distributed to all users.

See the “Header files” article in the *Encyclopedia* for information about standard system header files that are frequently called in user code. Individual header files each have an *Encyclopedia* article that describes their contents and uses.

Error handling

All programs should be coded to handle any errors that could arise; failure to do this can corrupt your data. Errors can result from user errors or system failures. For the purposes of this discussion, we will consider an error to be anything that happens that should not have happened.



Chapter 3

Common questions

This chapter lists some of the most commonly asked questions received by the Support Department from customers of earlier versions of the Development System.

Getting help

Q: What information should I have ready when I call SCO for help with a problem?

A: You should have the following information ready:

- Which release of the SCO Development System are you using?

To determine the release, type:

```
grep rel= /etc/perms/soft
```

- Which release of the operating system are you running? If you are using SCO XENIX, determine the release by typing:

```
uname -r
```

If you are using SCO UNIX System V/386 or SCO Open Desktop, determine the release by typing:

```
swconfig
```

(Log in as root to do this.) Search for the line that reads "SCO System V Operating System" or "Open Desktop".

- What is the exact error message?
- What were the exact commands that were being used? If a program is being written, what are the exact lines of code that are in question? Do you have a short test program that will demonstrate the problem?

Installing

Q: Can I install the Development System if it comes on 5-1/4" disks and the Operating System came on 3-1/2" disks, or vice versa?

A: In general, we recommend that you use the same media for installing the Development System as for installing the operating system. The **-m** option to **custom(ADM)** directs **custom** to look at a device other than the primary floppy drive. Because operating system disks must be used when installing the Development System, it is extremely difficult to utilize this option.

Q: During reinstallation of the SCO UNIX System V/386 Development System, I get the following error from **custom**:

```
"Error: Cannot open perm file"
```

A: **custom** removed the file `/etc/perms/dsmd` when the SCO UNIX System V/386 Development System was removed. During reinstallation, **custom** doesn't reinstall this file. The file `/etc/perms/dsmd` must be reinstalled manually before attempting to reinstall the SCO UNIX System V/386 Development System using **custom**. To do this, perform the following steps:

1. Log in as **root** and type:

```
grep 'PERM.*etc/perms/dsmd' /etc/perms/* | awk '{print $6}'
```

This outputs a disk number. If the disk number is N01 or N02, then proceed to step 2, otherwise perform the following:

a. Insert the disk with the number from step 1 into your primary drive.

b. Type:

```
cd /
```

c. Type:

```
tar xvf /dev/rfd096ds15 ./etc/perms/dsmd
```

d. Type:

```
cp /etc/perms/dsmd /etc/perms/dsmd.bak
```

e. You are finished; do not perform step 2.

2. Perform the following steps:

- a. Mount the N1 disk by typing:

```
mount -r /dev/fd096ds15 /mnt
```

- b. Type:

```
cp /mnt/etc/perms/dsmd /etc/perms/dsmd
```

- c. Type:

```
umount /dev/fd096ds15
```

- d. Type:

```
cp /etc/perms/dsmd /etc/perms/dsmd.bak
```

The next time the SCO UNIX System V/386 Development System is removed, you can simply type:

```
cp /etc/perms/dsmd.bak /etc/perms/dsmd
```

This restores it; then proceed with the reinstallation.

Q: I am attempting to install the SCO UNIX System V/386 Release 3.2 Development System Version 2.0, but when I get to the last diskette where the serialization occurs, the installation fails. What is the problem?

A: The most likely cause of the problem is the release of the SCO UNIX System V/ 386 Operating System. Confirm the release by typing

```
grep rel= /etc/perms/inst
```

SCO UNIX System V/386 Release 3.2 Development System Version 2.0 requires SCO UNIX System V/386 Release 3.2 Operating System Version 2.0 to install successfully. An upgrade is available and may be obtained by contacting the SCO Sales Department, if you are currently running SCO UNIX System V/386 Release 3.2 Operating System.



- Q:** After installing the SCO UNIX System V/386 Development System, creating a user using `sysadmsh(ADM)` fails with the following messages:

```
useshell: File Control database inconsistency
useshell: Not authorized
```

- A:** Installation of the SCO UNIX System V/386 Development System alters the ownership and group of the `/usr/lib` directory. Therefore, the security subsystem will not allow access to the scripts contained in `/usr/lib/mkuser`. These scripts are invoked by `sysadmsh(ADM)` when creating a new user. You must log in as `root` and modify the ownership and group of the directory `/usr/lib` as follows:

```
# chown bin /usr/lib
# chgrp bin /usr/lib
```

You should now be able to invoke `sysadmsh(ADM)` successfully and create new users.

- Q:** I am attempting to install the SCO UNIX System V/386 or SCO XENIX System V Development System. Near the end of the installation, I am prompted to insert a diskette from the corresponding operating system. My Development System media is on 3-1/2" diskettes and my operating system media is on 5-1/4" diskettes. What do I do?

- A:** Installation of the SCO UNIX System V/386 or SCO XENIX System V Development System requires that an operating system diskette be inserted. The installation of the Development System cannot succeed unless the media types for the Development System and the operating system are the same. If the Development System media is different from the media of the operating system, return the Development System to the distributor to be exchanged for the same media as the operating system.

- Q:** I want to install the Development System using the secondary floppy diskette drive. When I do, it works fine until the system asks for the N diskette. When the diskette is inserted in the second drive and the <Return> key is pressed, the system accesses the primary diskette drive. How do I install the Development System?

- A:** The Development System must be installed using the primary floppy diskette. Switch the drives around and install the Development System again.

Preprocessing

Q: Why does `/bin/cc` expand my macro incorrectly?

A: The proposed ANSI standard for C does not allow the substitution of macro formal parameters within quoted strings. However, many existing C compilers allow this. Consider the following program fragment:

```
#define CTRL(c) ('c' & 0x1f)
putchar(CTRL(g));
```

The statement operates as intended when using the AT&T Compiler (`gcc`). The AT&T compiler expands this to:

```
putchar(('g' & 0x1f));
```

But, according to the ANSI standard, it should be expanded to:

```
putchar(('c' & 0x1f));
```

The Microsoft C compiler follows the ANSI standard strictly; therefore, you must define and use the macro as follows:

```
#define CTRL(c) (c & 0x1f)
putchar(CTRL('g'));
```

The compiler expands this to:

```
putchar(('g' & 0x1f));
```

Q: Why does a compiler assertion error occur when compiling using the `-Oi` or `-Ox` flags?

A: The flag `-Oi` tells the compiler to expand certain intrinsic functions inline rather than making function calls to them. The `-Ox` flag is equivalent to `-Oactli`, which includes the `-Oi` flag. The compiler is failing to work correctly when it comes across intrinsic functions. To prevent this assertion error, you can either avoid using `-Oi`, or you must use either an *intrinsic pragma* or a *function pragma* to generate intrinsic functions. For example, to instruct the compiler to use inline code for the `pow(S)` function, you must add the following line to your code:

```
#pragma function(pow)
```



The following program illustrates this:

```

/*
** foo.c:                Demonstrates Intrinsics compiler Problem
**
**      cc -c -Oi foo.c    OR      cc -c -Ox foo.c
**
** We get:

Compiler error (assertion): file @(#)grammer.c:1.32, line 118 source=6

** The "#pragma function" below will instruct the compiler to generate
** inline code for pow. Uncomment this line if you wish to use pow as
** an inline function. Uncommenting this line will stop the assertion
** error.
*/

/*
#pragma function(pow)
*/

g() {
    double factor, pow();
    short sh;

    factor = pow(1.0, (double) sh);
}

```

- Q:** I am using the **-E** (or **-P**) option to preprocess my source file. The program initializes a large (over 100) static character array. In the generated code file, newline characters are inserted in various points within the static character array. When I try to compile the preprocessed file, the compiler gives syntax errors. Why?
- A:** This problem may be avoided by changing the style with which long strings are split over multiple lines. The *K&R compatible* string splitting technique uses backslashes to escape newline characters. The split lines are rejoined by the C preprocessor. With SCO's ANSI-compliant cc compiler, it is also possible to use ANSI-compatible implicit string concatenation. The ANSI standard specifies that adjacent quoted strings are automatically turned into one longer string by the compiler. Thus, the following examples are equivalent:

```

char *longstring="This is split\
over two lines";

char *longstring="This is split"
" over two lines";

```

Note that in the first example, the number of spaces at the left of the continuation line is significant; in the second, it is not.

In some cases, the standalone preprocessor, `/lib/cpp`, may also be useful. `/lib/cpp` is not ANSI-compliant and has limited buffers for certain compile-time structures such as `#define` statements.

Q: Why can't I get `getc` and `putc` to work when I use `rcc`?

A: The header files were written to work specifically with the Microsoft C compiler. The `cc` compiler sets a number of `#define` statements by default. Some of the header files depend upon these defines. Unfortunately, the `rcc` compiler does not set them by default. The header file `stdio.h` needs to have `i386` defined so that `getc` and `putc` work correctly. To compile a program using `rcc`, the command line should look like this:

```
rcc -Di386 your_program.c
```

Q: Why does compiling a program with the header file `/usr/include/sys/proc.h` produce syntax and/or preprocessor errors?

A: The `proc.h` file contains references that are resolved within four additional header files. These header files should be, but are not, referenced within `proc.h`. Compilation can generate errors similar to the following:

```
/usr/include/sys/proc.h(43) : error C2061: syntax error : identifier 'uint'
/usr/include/sys/proc.h(43) : fatal error C1022: expected '#endif'
```

The exact errors generated differ depending on what other header files are contained within the specific program. It is necessary to edit `/usr/include/sys/proc.h` and insert the required header files, which will resolve all references. Shown here are the `#include` references before and after the addition of the necessary files:

BEFORE:

```
#include <sys/signal.h>
#include <sys/ci/cilock.h>
<REST OF /usr/include/sys/proc.h DELETED>
```

AFTER:

```
#include <sys/signal.h>
#include <sys/ci/cilock.h>
/* THESE ARE THE FILES TO BE ADDED */
#include <sys/types.h>
#include <sys/immu.h>
#include <sys/region.h>
#include <sys/param.h>
<REST OF /usr/include/sys/proc.h DELETED>
```

Alternatively, you can just add the four header files to your main program before you include the *sys/proc.h* file as follows:

```
/* This program won't compile w/o the following include files. */
#include <sys/types.h>
#include <sys/immu.h>
#include <sys/region.h>
#include <sys/param.h>
#include <sys/param.h>
main()
{
    /* rest of program */
}
```

Q: Why can't I compile a program preprocessed with **cpp**?

A: If you run **/lib/cpp** without the **-P** flag, it creates badly formed line directives. The **cc** compiler then fails with an error message such as:

```
hello.c(1): error C2019: expected preprocessor command, found '1'
```

The documentation for **cpp** warns that its use, other than through the **cc(CP)** command, is not suggested. You can successfully preprocess a source file with proper line directives using the **-E** flag to **cc**. You can also use the **-P** to **cpp**, as it omits the line control directives.

Compiling

Q: When I link object files to create a XENIX binary, it works fine. Why do I get an undefined symbol message from **ld(CP)** when I try to link object files to create a UNIX binary?

A: The UNIX compilers require that all libraries specified on the **cc(CP)** or **rcc(CP)** line be named with the **-l** option to the right of the C source filename. For example, the following **cc** line is correct; externals in *main.c* are resolved by the math library:

```
cc main.c -lm
```

The compiler on the XENIX operating system silently moves all **-l** specifications to the end of the line if necessary, and so would compile correctly if the **-l** option were specified to the left of the C source filename.

Q: What does the compiler message `Unresolved Externals` mean?

A: This message indicates that the compiler could not locate all functions that are called from within the program. Usually this means one of the following:

- You used a function that is in a library that was not specified with a `-l` on the `cc` line. Check the section (S) manual pages for all functions used in the program to see which `-l` options are required.
- You specified `-l` to the left of the C source filename. The UNIX compilers require that the `-l` option be to the right of the source filename.
- Routines may have been added to the archive since `ar(CP)` or `ranlib(XNX)` was run on the archive.

Q: How do I make `cc(CP)` and `ld(CP)` look in a different directory for libraries?

A: The following flags can be passed from `cc` to `ld`:

- `-BL directory` tells `ld` to look in *directory* for special modules (`crt1` and `crti`)
- `-Y L, directory` tells `ld` to look in the *directory* for the libraries specified (`-lc`, `-lm`, and so forth)

For example, the following command line tells `ld` to look in the `/tmp/lib` directory for the special modules and whatever libraries are specified on the compile line:

```
cc -Y L,/tmp/lib -BL /tmp/lib file.c
```

`ld` also accepts the following flags:

- `-B1 file` for an alternate pass1 of the compiler
- `-B2 file` for an alternate pass2 of the compiler
- `-B3 file` for an alternate pass3 of the compiler
- `-pC file` for an alternate `cvtomf` converter

Q: Is it possible to create an OMF binary with the `rcc` compiler?

A: No, the `rcc` compiler can only create COFF binaries. If you need to create an OMF binary, use the Microsoft C compiler with the `-xenix` or `-x2.3` flag. For example:

```
cc -x2.3 foobar.c
```

- Q:** When I am trying to compile a program which has 15 or more nested loops in it, the compile fails with the message:

fatal error C1026: parser stack overflow, please simplify your program

How can I avoid this?

- A:** The Microsoft C compiler provided with SCO UNIX System V/386 and SCO XENIX Development Systems can only handle nests of up to 14 loops. To work around this, you must simplify your program by breaking the loops down into separate functions. If you have a version of the SCO UNIX System V/386 Release 3.2 Development System, you can work around this problem by compiling the offending object modules with the AT&T compiler, `rcc`.

- Q:** The SCO UNIX System V/386 Development System Release 3.2 C compiler is only printing an error number but no comment follows. There are no missing file messages. What is needed to get the compiler to print the error messages with the error number?

- A:** There are files that the compiler is using to print the error messages. These files are either not installed or they are corrupted. The following files must be present for the compiler to print error messages:

/lib/386/c1.err
/lib/386/c23.err
/lib/c1.err
/lib/c23.err

The easiest way to install these files is with the **custom** (ADM) utility. It is also possible to use **grep** for the files listed above in the file `/etc/perms/soft`. For example:

grep "[13].err" /etc/perms/soft <RETURN>

This allows you to find out which package the files listed above belong to. The files `/lib/386/c1.err` and `/lib/386/c23.err` belong to the 386XDEV package. The files `/lib/c1.err` and `/lib/c23.err` belong to the SOFT package.

- Q:** What do the five **-FP** flags listed on the `cc(CP)` manual page mean?

- A:** The **-FP** options to the C compiler are used when generating programs targeted for MS-DOS. These five flags control how the resulting program performs floating-point operations. Here are the meanings of these flags:

-FPa generates subroutine calls to the "alternate" floating-point library. This library is faster than the standard coprocessor emulation library, but not as accurate.

- FPc** generates subroutine calls (as opposed to inline code) to the coprocessor emulation library. Programs compiled with this flag will do all floating-point operations in software.
- FPc87** generates subroutine calls to the floating-point library, which then uses the coprocessor. A coprocessor must be installed to run programs compiled with this flag.
- FPi** generates inline code that checks to see if a coprocessor is present, use it if one is there, or call the emulation library if one is not. This is the default for DOS, and is the only method used for performing floating-point operations under XENIX.
- FPi87** generates true inline code for the coprocessor. A coprocessor must be installed to run a program compiled with this flag.

Again, note that programs compiled with **-FPa** or **-FPc** ignore a coprocessor if one is installed, and programs compiled with **-FPc87** and **-FPi87** do not run if a coprocessor is not installed.

- Q:** The **-Fo** option of the C compiler is not creating the object name specified. Instead, the object name of the file created is defaulting to the same name as the source file but ends with the **.o** suffix.
- A:** The **-Fo** option must appear before the source file in the command line. Move the **-Fo** option so that it appears before the source filename in the command line. For example, if your **cc** command currently appears as:

```
cc -c Foobar.c -FoBarfoo.o
```

It should be changed to:

```
cc -c -FoBarfoo.o Foobar.c
```

Also note that there is no space between **-Fo** and the object file name.

Linking

- Q:** I am having difficulty linking and find that three different versions of **ld** exist. Which shall I use? How are **/bin/ld**, **/lib/xout/ld**, and **/lib/coff/ld** related?
- A:** Both **cc** and the AT&T compiler **rcc** use **/bin/ld** when linking object modules. **/bin/ld** is a front-end to the system linkers **/lib/xout/ld** and **/lib/coff/ld**. If the object module to be linked is COFF format, then **/bin/ld** calls **/lib/coff/ld** to perform the link. Likewise, **/bin/ld** calls **/lib/xout/ld** for OMF format object modules. The SCO UNIX System V/386 Development System allows the creation of COFF (default) or OMF binaries when using **cc**. Only COFF binaries are created using the compiler **rcc**. OMF binaries

are created by adding the flags `-x2.3`, `-xenix`, or `-xout` to the `cc` compile line. Other Development System utilities have front-ends that access the corresponding COFF version located in the directory `/lib/coff` or the OMF version located in the directory `/lib/xout`. These utilities are `ar`, `nm`, `prof`, `size`, and `strip`. In cases where the format of the object module (`*.o`) is unknown, one may use the utility `file` to determine this information. Typing `file obj_name.o` shows the following:

```
(COFF)      obj_name.o:      iAPX 386 executable not stripped - version 30821
(OMF)      obj_name.o:      8086 relocatable (Microsoft)
```

Q: I am getting the error:

```
file.c
/bin/ld : fatal error: file.o : invalid numeric value
```

This same `cc(CP)` command worked in the SCO XENIX Development System Release 2.2.1 and earlier. What is wrong with it and why doesn't it work now?

A: In early releases of the Development System, `cc(CP)` was able to interpret the `-SEG512` option as `-SEG 512` (notice the space between `-SEG` and `512`). In later releases, the `cc(CP)` command line parser was modified such that it expects to see `-SEG 512` instead. The compiler no longer recognizes `-SEG` unless there is a space separating it from the number part of the option. The compiler, therefore, does not translate `-SEG` into the `-S` option when it calls `ld(CP)`. Instead, it passes `-SEG512` directly to `ld(CP)`. The loader then complains with an error.

Q: How do I compile several `.c` files into one `.o` file? I am trying to create a `Driver.o` file that can be linked into the kernel.

A: The link editor creates a single `.o` file from several `.c` files. Creation of a `Driver.o` to be linked into the kernel is the only instance in which the link editor should be called from the command line. Usually, it is called using the command `cc(CP)`. Use the following commands to create `Driver.o` from `file1.c` and `file2.c`:

```
cc -c file1.c
cc -c file2.c
ld -r -M3s file1.o file2.o -o Driver.o
```

For additional information, the directory `/usr/lib/samples/pack.d` contains several subdirectories of sample drivers. Each subdirectory has a makefile demonstrating compilation of the directories' contents.

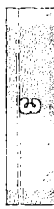
- Q:** When attempting to link 386 object modules from a library into a program, the linker returns an error message of unknown record type and/or symbol table read error. What does this mean?
- A:** `/bin/ld` does not understand 386 object module format. Use `/usr/bin/ld386`, which does understand the new format.
- Q:** When I link object files to create a XENIX binary, it works fine. However, I get an undefined symbol from `ld(CP)` when I try to link object files to create a UNIX binary. Why?
- A:** There is a subtle difference in the `ld(CP)` programs provided with the SCO UNIX System V/386 and SCO XENIX System V Development Systems. For example, in SCO XENIX, unresolved externals in an object file are satisfied by any library that appears in the `cc` line. SCO UNIX System V/386 is different. Libraries satisfy unresolved externals only in those object files that occur to the left of them in the `cc` line. For example:

NOTE These examples assume the use of the SCO UNIX System V/386 Development System.

```
+----- externals in main will be resolved by
|           the math library
|
# cc main.c -lm

+----- externals in main will NOT be resolved by
|           the math library
|
# cc -lm main.c
```

Thus, a `cc` line that worked with SCO XENIX might not necessarily work when you recompile for SCO UNIX System V/386.



Q: When I compile with `-lmalloc`, I get an error message saying that `_calloc` is redefined. Why?

A: When compiling with the `-lmalloc` library and the `-lcurses` and `-ltermcap` libraries, you get the above error because one of the termcap functions calls an undocumented library support routine directly. This routine is `cfree()` and is defined inside of `calloc.o` in `libc.a`. The link editor, in trying to resolve the reference to `cfree()`, brings in the `calloc.o` object module, sees the `calloc()` label, and covers your screen with error messages. As a work-around, you can add the following function to your program:

```
cfree(p, num, size)
char *p;
int num, size;
{
    free(p);
}
```

This resolves the reference to `cfree()`.

Debugging

Q: Why won't `nm` sort symbols in large executables?

A: `nm` is a small model, and it runs out of space. Instead of using the `-n` option to sort a long name list, use the command line:

```
nm | sort
```

Q: Why won't `sdb` debug small model programs?

A: `sdb` only works on split I & D executables. To use `sdb` on small model programs, compile them with the `-i` option to `cc`. Medium, large, and huge model programs automatically use split I/D code.

Q: I am using CodeView to debug a program that contains lines greater than 78 characters. Why am I unable to see anything beyond the 78th character.

A: There is no line wrapping in CodeView. To view the portion of the line that exceeds 80 characters, perform a shell escape and `more`, or edit the file using an SCO UNIX System V/386 line editor such as `vi`. While in the dialog window, type either of the following:

```
!more filename.c
!vi filename.c
```

CodeView currently depends on a one-to-one mapping of screen lines to source code lines. Because most of the line is visible, there is usually enough information to do normal debugging.

For lines that contain a lot of indenting with tab characters, you can set the tab size to something smaller with the # command. To do this, enter the dialog window and type:

#<number>

number is the new number of characters for each tab character. In window mode, the screen is redrawn with the new tab width when you enter the command.

Q: I am doing mixed language programming with the SCO UNIX System V/386 Development System. Can I use CodeView to debug both C and FORTRAN modules?

A: CodeView may be used to debug C, FORTRAN, or BASIC programs. However, not all FORTRAN or BASIC compiler and linker versions support CodeView options. If you try to debug an executable file that was not compiled and linked with CodeView options, or if you use a compiler that does not support these options, then you may only be able to use the debugger in assembly mode. This means that the CodeView debugger does not display source code or understand source-level symbols, such as symbols for functions and variables.

Q: Why doesn't CodeView run in window mode when run on a serial terminal, or in a SCOTerm or Xterm window?

A: CodeView is a graphical based program that expects to talk to the video card directly. When it runs on a serial terminal, it runs in sequential mode because this device does not use a video card. When it is run in a SCOTerm or Xterm window, it must also run in sequential mode so it does not conflict with the X server, which is already talking to the video card.

NOTE Note that SCOTerm runs a character-based application in the same manner that the application works on the ANSI console. As CodeView is a graphics-based application, it cannot be run in a SCOTerm window in window mode.

Q: Does the **masm** debugger include debugging information in the object modules?

A: Like **cc**, **masm** understands the **-Zi** flag when compiling the assembly language source files. When running **masm**, use the **-Zi** flag if you want debug symbols to be preserved. For example:

```
masm -Zi foo.s
```

This produces a *foo.o* file with debug symbols included.

It is still necessary to execute the final link in such a way that the debug symbols are preserved. If you wanted to link the previously created file *foo.o* with the C startup and runtime libraries and preserve the symbolic information, you could link it like this:

```
cc -Zi foo.o
```

Q: Why do I get the following error message when I run **dbXtra**?

```
dbxtra: fatal error: filename too long
```

A: To fix this problem, it is necessary to rename any source files, where their basename parts are 10 characters or longer, into shorter names. For example, *LongFileName.C* should be renamed to *LongFName.C*. The error message occurs because **dbxtra** tries to open *filename.xref*. A filename of 10 characters or more results in a filename of 15 characters which is illegal. If the file is renamed to a name of 9 characters or less, the error message will no longer occur.

Functions

Q: How do I define a function with a variable number of arguments?

A: Use the macros documented on the **varargs(S)** manual page to write portable functions that accept a variable number of arguments. Using function prototypes enables the compiler to check that each function is getting the correct arguments, which simplifies the debugging process. See the "Functions" article in the *Encyclopedia* for more information.

Q: Why should you declare functions before using them?

A: If a function is not declared, the compiler treats the function as an integer. Integer sizes vary depending on the memory model utilized and the machine architecture. If a function that returns a double is not declared, the compiler treats this as an **int**. In the example below, if **drand48()** is

undefined, the non-negative double-precision value returned fails, because only 16 or 32 bits have been allocated for storage. Declaring the function as a double before use is required to ensure proper handling. This is a specific case for why one must always declare functions before using them:

```
main()
{
    double r, drand48();          /* if drand48() is not declared
                                   a floating exception error
                                   will occur */

    int x;
    long seed=1.0;
    void srand48();

    srand48(seed);
    for (x=0; x < 20; x++)
    {
        r=drand48()
        printf("Random number is %g0,r);
    }
}
```

Q: Why should I use function prototypes? What are they?

A: A function prototype is a declaration of a function where the types of parameters are explicit and are part of the type of the function. Here is a contrived example:

```
float goober(float x)
{
    return (x + 1.0);
}
```

Specific rules have been set up about prototypes and passing arguments to functions. If a function prototype does not appear in the code before it is used, nothing is assumed about the function arguments. In the following program, the compiler does not complain about the fact that we are passing a pointer to float to **goober**.

```
main()
{
    float y = 5.0, *p = &y;

    /* This is wrong we meant to pass *p instead of p. */

    printf("The value is %f0, goober(p));
}
```

Similarly, if a function declaration does not include arguments, no assumptions are made about these arguments. In the following example, **goober** has been declared in this manner. Again, the compiler has no complaint about passing a float pointer.

```
main()
(
/* The compiler still doesn't know what arguments to expect. */
    float goober(), y = 5.0, *p = &y;

/* This is wrong we meant to pass *p instead of p. */
    printf("The value is %f0, goober(p));
)
```

If we include the float prototype before calling **goober**, the compiler warns us about this type mismatch. In actual code development, prototyping can help you check to see if each function is getting the correct arguments and speeds up the debugging process.

```
/*
To compile: cc file.c -lm
Two warning messages should come back:
file.c
file.c(5) : error C2115: 'argument' : incompatible types
file.c(5) : warning C4024: 'goober' : different types : parameter 1
*/

main()
(
    float goober(float), y = 5.0, *p = &y;

/* This is wrong we meant to pass *p instead of p. */
    printf("The value is %f0, goober(p));
)
```

Q: The function `toa()` is listed under the DOS services section and converts long integers to characters. Is there a counterpart for SCO UNIX System V/386 and SCO XENIX?

A: The function to use when developing programs for the SCO UNIX System V/386 or SCO XENIX environment is `sprintf()`. The following example is a program that uses `sprintf()` to convert a long to a null-terminated character string:

```
#include <stdio.h>

main()
{
    long x;
    char string[10];

    x=40;                      /* assign a long value */
    sprintf(string, "%d0,x);    /* Assigns the value of x to string */
    puts(string);              /* Prints the string to stdout*/
    exit(0);
}
```

For additional information on `sprintf(S)`, refer to the *SCO UNIX System V/386 Development System Programmer's Reference*.

Q: How do I use `setjmp(S)` and `longjmp(S)` in a program?

A: The `setjmp` system call saves a copy of the program's execution state. The `longjmp` system call changes the current execution state to the previously saved state. After `longjmp` completes, program execution continues as if the corresponding call of `setjmp` had just returned.

At the time of the second return from `setjmp`, static and external variables have the same value as before the call to `longjmp`. The value of automatic and register variables are undefined and nothing should be assumed about their contents.

`setjmp` returns 0 when it is first called. `longjmp` causes `setjmp` to return with a non-zero value. This value can be set from `longjmp`. We can differentiate between the two situations depending upon the return value from `setjmp`.



setjmp and **longjmp** return a program to a known state after handling a signal. For example:

```
/*
   To compile: cc file.c
   Program will finish executing when i equals 10.
*/

#include <signal.h>
#include <setjmp.h>
#include <stdio.h>

jmp_buf sjbuf;
int i;

main()
(
    int onintr();

    if (!setjmp(sjbuf)) /* save the process' execution state */
        printf("first return from setjmp0);
    else
        printf("returning after lngjmp0);

    signal (SIGINT, onintr); /* trap the interrupt signal */

    i = 0;
    while (i != 10) {
        printf("i = %d0, ++i);
        sleep(3);
    }

    int onintr()
    (
        printf("0nterrupt...resetting execution state0);
        longjmp(sjbuf, 1);
    )
}
```

Q: How do I write a function in the C language that takes multiple arguments?

A: As an example, the following program takes several arguments. All of the arguments are character pointers, but they do not need to be. **printf** uses the first string to determine what is following it. This routine assumes character pointers until it hits a **NULL**.

```

#include <stdio.h>
#include <varargs.h>

main()
{
    My_Func( "hello", "1", "3", "5", NULL );
}

My_Func( va_alist )
va_dcl
{
    va_list pa;
    char *p;

    va_start( pa );

    /*
    **      va_arg takes the pointer set up with va_start,
    **      and the size of the particular element that one
    **      expects next.
    **
    **      One could restart the search by calling va_start again.
    */

    while( ( p = va_arg( pa, char * ) ) != NULL ){
        printf("%s ", p );
    }
    putchar('\0');
    va_end( pa );
}

```

Libraries

Q: Where do I create my own library so the compiler can find it?

A: The steps below create both a 286 and 386 Development System library.

1. Compile the module(s) to be incorporated into your library using the command:

```

cc -M3s -c file.c
cc -M2l -c file.c

```

The first compile line produces an executable for the 386 small model. The second compile line produces an executable for the 286 large model. Both compile lines produce an object file named *file.o*.

2. Create an archive that includes the object file (or files) from the first step, using one of the following commands:

```
ar qcv /lib/Llibnameit.a file.o    (286)
ar qcv /lib/386/Slibnameit.a file.o (386, XENIX)
ar qcv /lib/libnameit.a file.o     (386, UNIX)
```

If the library is not to be added to the system libraries but is intended for individual use, type:

```
ar qcv /usr/login_name/lib/userlname.a file.o
```

3. For OMF archives, run **ranlib**(XNX) on the library. If you do not do this, attempts to link with the library fail.

```
ranlib /lib/386/Slibnameit.a    (386)
ranlib /lib/Llibnameit.a        (286)
ranlib /usr/login_name/lib/userlname.a (User lib)
```

ranlib must also be run after an existing library is modified.

NOTE Do not use **ranlib** if you are creating a COFF library.

4. If including system libraries, the newly created library can now be linked during compilation of a program by using the command:

```
cc -M3s main.c -lnameit          (386, XENIX or UNIX)
cc -M2l main.c -lnameit          (286)
```

To link a user-defined library, use the command:

```
cc -M3s main.c /usr/login_name/lib/userlname.a (386)
cc -M2l main.c /usr/login_name/lib/userlname.a (286)
```

NOTE The portion of the library name following “Slib” (for example, “nameit”) in the archive */lib/386/Slibnameit.a* is the name used in step 4 with the **-l** option.

For additional information, see the **cc**(CP), **ranlib**(XNX), and **ar**(CP/XNX) manual pages.

Q: When creating a library, why does **ranlib** report lib*.a not an archive?

A: **ranlib**(XNX) is only required to be run on OMF libraries. You are not required to run **ranlib** on COFF libraries.

The UNIX Development System can create both OMF and COFF object modules. If you are creating object modules using the compiler flags **-x2.3**, **-xenix**, or **-xout**, you must run **ranlib** on the library you generate. If you are not using these flags, you are creating COFF modules. The resulting library you create does not need to have **ranlib** run on it.

Attempting to use **ranlib** on a COFF library produces the following error message:

```
lib*.a: not an archive
```

Use the **file(C)** command to determine if an existing library is in COFF or OMF format. COFF and OMF may not be mixed when combining object modules or incorporating libraries. Attempting to create a library that contains some object modules compiled OMF and others compiled COFF fails with the message

```
ar: testdir.o not in archive format
```

Attempting to link COFF and OMF object modules with **cc** results in the error:

```
Cannot link both coff and omf records. Exiting.
```

Most source code can be compiled into either an OMF or COFF binary. If you have OMF routines that must be used with COFF routines, recompile them into COFF binaries.

Users should not use the **cvtomf** and **cvtofff** utilities to convert binaries to the other format. The “COFF” article in the *Encyclopedia* explains the difficulties this can cause. For more information, see the **cc(CP)** and **file(C)** manual pages.

Q: Is there a routine that converts time from a string into a **long** that represents the number of seconds since 00:00:00 Greenwich Mean Time (or Universal Coordinated Time)?

A: This routine is not available in the current release.

Q: Where are the crypt libraries?

A: United States law does not allow us to ship the crypt libraries to locations outside the United States. Domestic customers may request a copy of the crypt libraries from The Santa Cruz Operation Media Department at no additional charge.

Q: The math routines documented under **trig(S)** do not resolve. **/bin/ld** fails with errors about undefined symbols for all the **trig(S)** math routines. What library do I link with?

A: If you use the **trig(S)** math routines, the **-lm** to the C compiler must be specified. The routines documented in **trig(S)** are:

sin
cos
tan
asin
acos
atan
atan2

Q: Why do I get the following error when I run my program?

"fltrd: Floating Point Not Present"

A: When using library functions that point to a floating-point number variable, such as **sscanf()**, you must ensure the variable itself is used within the source code to instruct the C compiler to include the floating-point library. The following code segment illustrates this usage:

```
/*
** cc -DFIX foo.c -o foo    and run "foo".  The error message
** "fltrd: Floating Point Not Present" will no longer occur.
**
** cc foo.c -o foo          and run "foo" to see the error message.
*/
char buffer[] = {"100"};

main() {
    float d[10];
    int i[10];

    sscanf(buffer, "%f", &d[0]);
    sscanf(buffer, "%f", &i[0]);

#ifdef FIX
    printf("d=%f", d[0]);          /* Use the floating-point variable */
#endif

    printf("i=%d", i[0]);
    printf("%s", buffer);
}
```

Q: The manual states that `ld` only uses routines needed to resolve externals. However, more routines are incorporated into my executable than should be. Why is my linked file much larger than I anticipated?

A: A library may consist of several object modules, each of which contain many functions. When a program needs to resolve an external function at link time, the object module from the library containing the function reference is incorporated, not just the routine within the object module. Therefore, other functions contained in that object module are also incorporated, although they may not be referenced by the program.

Q: What does the message `Unresolved Externals` mean? How can I determine which library contains a certain function?

A: When a program uses functions, and the library that contains the text of these functions is not included on the compile line, this message is generated. This message can also occur if routines have been added to the library but the library has not been updated using `ar` and `ranlib`.

To determine which library contains the text of a function, first check the manual page for that function. The section labeled "Notes" should list the library that needs to be included on the compile line. As an example, the `sqrt` function requires the header file `math.h` and the library `/lib/386/Slibm.a` when compiling an 80386 OMF binary. The program that contains `sqrt` would require the compile line:

```
cc filename.c -lm
```

If you cannot determine which library contains the function text, you may use the following commands either to determine what directories contain libraries (`*.a`) or which library contains the function text. The `find` command can be used to output all files with an extension of `.a`:

```
find / -name *.a -print > /tmp/libfile
```

To search through all `*.a` files found, type the following:

```
sh <Return>    - Must be running a Bourne Shell
for i in `cat /tmp/libfile`
do
nm $i | grep function_name && /bin/echo " 07Library - $i"
done
```

`function_name` is the name of the function you are searching for. For example, `sqrt`. To determine which library directories to search, use the following guidelines:

- `/lib` compiles 8086 and 80286 memory models to create OMF binaries. The library is preceded by a C, L, M, or S for compact, large, middle, and small memory models.

- `/lib/386` compiles 386 small model OMF binaries. The library name is preceded by an S as in `Slibm.a`.
- The directories `/usr/lib` and `/usr/lib/386` may also contain additional libraries that should be searched for unresolved externals.
- If you are compiling under SCO UNIX System V, the default binary created for 80386 is a COFF binary and the libraries are located in `/lib`. To create OMF binaries use `cc -x2.3` or `-xenix filename.c`. COFF libraries are not preceded by a memory model designator. The COFF math library for 80386 is `/lib/libm.a`, whereas the OMF 80286 middle model counterpart is `/lib/Mlibm.a`

Q: Why can't I run AT&T or non-SCO binaries in the shared library, `libnsl_s`?

A: The `libnsl_s` source, when built with the Microsoft C compiler, produces a `libnsl_s` binary that is incompatible with the AT&T or ISC version of `libnsl_s`. The `Libnsl_s` binary was rebuilt using the AT&T C compiler and the program `chshlib` was used to verify that this library was compatible with the AT&T or ISC `libnsl_s`.

There are several key programs that are built with SCO versions of `Libnsl_s`, so SCO now ships an SCO-compatible version (`/shlib/libnsl_s`), and an AT&T-compatible version (`i/shlib/libnsl_s.att`) of `libnsl_s`.

Q: What is `libnsl_s.att` and how does it relate to the `libnsl_s`?

A: To run the AT&T-compatible version, you must first run the program `fixshlib` on the COFF binary. By using `fixshlib filename`, when executed, the COFF binary picks up the AT&T-compatible `/shlib/libnsl_s.att`.

Q: Why do I get the following error when I run a program that uses SCO ISAM library calls?

```
Error: Incorrect SCO ISAM Runtime System installed: 0
```

A: This error occurs when the file `/usr/lib/isam/locks` is not found. The file `/usr/lib/isam/locks` is created upon installation of the SCO ISAM package.

To recreate this file, log in as `root` and issue the command:

```
/usr/bin/isverify -I
```

NOTE `/usr/bin/isverify` is documented in the Appendix D the *SCO ISAM User's Reference*. The `-I` option creates the `/usr/lib/isam/locks` file, then issues the message:

```
SCO ISAM Runtime System initialization complete
```

If the directory `/usr/lib/isam` does not exist, it is created with correct permissions.

In SCO Open Desktop Release 1.1.0, **isverify** is shipped with ODT-OS. **isverify** is also shipped with the SCO Open Desktop Development System Release 1.0. It is not shipped with SCO Open Desktop Release 1.0.0, ODT-OS.

Q: When I compile 386 binary with `-Zp1` or `-Zp2`, then C library functions fail. Why do 386 C library functions such as `printf` not work properly when the `-Zp1` or `-Zp2` flags are used to modify data structure alignment?

A: 386 C library functions are compiled with `-Zp4` (that is, align structures on 4-byte boundaries). Therefore, any code that passes data to and from the C libraries in structures is expected to be compiled with `-Zp4`. Put the following into source code including C library header files compiled with `-Zp1` or `-Zp2`:

```
#pragma pack(4)
#include <stdio.h>
#pragma pack()
```

Q: I am trying to link the compiled object file of a database with a compiled object of a C program and another language. Why do I get one of the following error messages?

```
Unresolved Externals...
```

```
Symbol defined more than once: __routine_name in
file /lib/Xlibname.a (froutine_name.o)...
```

A: The message `Unresolved Externals...` indicates that a library, containing a function used in a program, is missing. The message `Symbol defined more than once...` indicates that the same function is defined in two different archives. The object module(s) that contains the function(s) needs to be removed from the archive.

The commands **nm** or **ar** may be used to view the archive or archives that appear to be in conflict. You need to extract the offending module from the library(s) and recompile. After first copying the existing library to another file, for example *library.old*, extract the module from the library by typing either of the following:

```
ar x /lib/Xlib_name.a froutine_name.o (System lib)
ar x /usr/lib/name.a froutine_name.o (User lib)
```

As the archive date changes, you need to run the following:

```
ranlib /lib/Xlibname.a
```

Now, you are ready to recompile your program using the modified libraries by typing one of the following:

```
cc -M3se file.c froutine_name.o -llib_name -ldb -lx .. (386)
cc -M2l file.c froutine_name.o -llib_name ... (286)
cc -M3s file.c froutine_name.o /usr/lib/name.a ...(User lib)
```

NOTE The object module must be before any library flags on the command line. The **-lx** option searches for XENIX-specific system calls.

The previous examples demonstrate modifying system libraries and user-defined libraries for 386 and 286 memory models. For additional information about the commands used, refer to **cc(CP)**, **ar(CP)**, and **ranlib(CP)** in your *SCO XENIX System V Development System Programmer's Reference*.

Manipulating the console and terminals

- Q:** When I try to draw a box around the screen while in "vt100" mode, the bottom right corner of the box is missing. Why?
- A:** The problem arises because the "vt100" terminal scrolls one line up if any character is written to position 23,79. This is inherent in the "vt100" terminals and any program that may emulate a "vt100". The following program demonstrates the problem.

```
/* Simple program to draw a box around the entire screen */
/* Compile the program with cc file.c -lcurses. */

#include <curses.h>

main ()
{
    initscr();
    box(stdscr,0,0);
    refresh();
    endwin();
}
```

The workarounds are as follows:

1. Echo escape sequences to put the terminal into protected mode and reset the SCROLLWIN bit in the flags word.
2. Don't use vt100 emulation, but instead use wy60 or some other terminal emulation.
3. Draw a smaller box that does not place a character on the 24th line.

The following sample program illustrates these points.

```
/* Here is an example program that will draw a box including
   position 23,79 . Compile with cc file.c -lcurses */

#include <curses.h>

main ()
{
    short flags;

    initscr();

    flags = stdscr->_flags;      /* save the current flag settings */
    stdscr->_flags &= 0xFFFF0;   /* zero out the SCROLLWIN bit */

    addch (' 33');              /* escape sequence for ALT CHAR mode */
    addch ('$');
    refresh();

    addch (' 33');              /* escape sequence for PROTECTED mode */
    addch ('&');
    refresh();

    box(stdscr,0,0);
    refresh();

    /* escape sequenced for exiting ALT CHAR mode and PROTECTED */
    /* mode respectively, although not needed for this example */
    /* addch (' 33');
    * addch ('%');
    * refresh();
    * addch (' 33');
    * addch (' 47');
    * refresh();
    */

    stdscr->_flags = flags;      /* restore the flags for stdscr */
    endwin();
}
```



```
/* Here is an example program that will draw a box which
ends on the 23rd line. Compile with cc file.c -lcurses */

#include <curses.h>

main ()
{
    WINDOW *win;

    initscr();
    erase();
    refresh();
    win = newwin(23,80,0,0);
    box(win,0,0);
    wrefresh(win);
    sleep(2);
    endwin();
}
```

Q: I have a VGA Video Adaptor that supports 132 columns and I would like to program the adaptor for this display mode. Is there an IOCTL to accomplish this?

A: There are no IOCTLs that can change the column display to 132 columns, even though the video adaptor has support for this. The SCO XENIX or SCO UNIX System V/386 console driver supports only 80 column displays at the operating system level.

Q: How do I generate sounds at the system console?

A: If your purpose is to attract the user's attention, simply write a bell character (octal 007, <Ctrl>G) to their tty. This works on ASCII terminals as well as the console, and produces the same note to which the user is accustomed. For example:

```
putchar(' 07'); /* Send an ASCII bell character to stdout */
```

If you need to control the PC sound generator with more precision, an `ioctl()` function is available. `KIOCSOUND`, defined in `sys/vtkbd.h`, turns the sound generator on with a specified frequency. A frequency of 0 turns it off. A good ear can hear frequency values between 125 and 30,000. Note that these frequency values are not in hertz.

Using curses

Q: Which **curses** library should I choose as my default, and what does this mean? What if I don't have the library */lib/libcurses.a*?

A: During the installation of the SCO XENIX 386 Development System, you are asked to choose a default terminal description library for use with the **curses(S)** screen handling package. You can choose **terminfo**, **termcap**, or you can choose to have no default package by selecting neither.

If you choose **terminfo curses**, the files */lib/386/Slibtinfo.a* and */lib/386/Slibcurses.a* are linked. If you choose **termcap curses**, the file */lib/386/Slibtcap.a* is linked to */lib/386/Slibcurses.a*.

If no default **curses** package was chosen, then you must manually link the appropriate files, as the directory */lib/386* does not contain the library *Slibcurses.a*. For **termcap curses**, this would be done as follows:

```
ln /lib/386/Slibtcap.a /lib/386/Slibcurses.a
```

See the **ln(CP)** manual page for more information.

Q: How do I read the function keys in **terminfo curses** to detect items such as shifted function keys?

A: The following program demonstrates how to read the function keys (F1) to (F10) using **terminfo curses**. You may need to edit the */usr/lib/terminfo/terminfo.src* file and run **tic(C)** on this file to define your function keys before running this program. For example, this is a sample *terminfo* file for an **ansi** terminal, with (F1) to (F10) defined:

Defining function keys in terminfo

```
ansi|Ansi standard crt,
    cr=^M, cudl=^J, ind=^J, bel=^G,
    ill=E[L, am, cubl=^H, ed=E[J, el=E[K,
    clear=E[2JE[H, cup=E[%i%p1%d;%p2%dH, cols#80,
    dch1=E[P, dll=E[M, cudl=E[B, cbt=E[Z, rmir=,
    home=E[H, ich1=E[@, smir=, lines#25,
    cuf1=E[C, ht=^I, smso=E[7m, rmso=E[m, smul=E[4m,
    rmul=E[m, cuul=E[A,
    kbs=^h, kcuul=E[A, kcudl=E[B, kcubl=E[D,
    kcuf1=E[C, eo, ind=E[S, ri=E[T,
    kf1=E[M, kf2=E[N, kf3=E[O, kf4=E[P, kf5=E[Q, kf6=E[R,
    kf7=E[S, kf8=E[T, kf9=E[U, kf10=E[V, kf11=E[W, kf12=E[X,
```

Use the following command to compile this program:

```
cc -M3s file.c -ltinfo -lx          (UNIX or XENIX 386 Dev Sys)
cc -F 2000 -M2l file.c -ltinfo -lx  (XENIX 286 Dev Sys)
```

```

1 #include <signal.h>
2 #include <tinfo.h>
3 main() {
4     int die();
5     int ch;

6     signal(SIGINT, die);
7     initscr();
8     keypad(stdscr, TRUE);
9     scrollok(stdscr, TRUE);
10    noecho();
11    addstr("Type 'q' to quit");
12    refresh();
13    for(;;)
14    {
15        ch = getch();

16        if (ch == 'q') break;
17        if ( ch > 255 )
18            printf("Function key pressed was %d, (ch - KEY_F(0))");
19        else
20            printf("character was '%c', (ch);
21        refresh();
22    }
23    die();
24 }

25 die() {
26     endwin();
27     exit(0);
28 }
```

Line 6	die gracefully if user pushes the key
Line 7	initialize screen processing
Line 8	enable function and keypad keys mapping
Line 9	allow terminal to scroll
Line 15	wait for a key
Line 26	restore the terminal

Q: `curs_set()` consistently returns `ERR`. How do you use `curs_set`?

A: The `curs_set()` routine returns `ERR` unless certain capabilities are defined in the terminfo or termcap file. `curs_set()` requires a single parameter of 0, 1, or 2 as follows:

`curs_set(0)` make the cursor invisible
`curs_set(1)` make the cursor very visible
`curs_set(2)` return the cursor to normal visibility

The success of the `curs_set()` routine is dependent on the following three capabilities being defined for the user's terminal in the corresponding terminal description file:

Curses routine	Termcap variable	Terminfo variable
<code>curs_set(0)</code>	<code>vi=</code>	<code>civis=</code>
<code>curs_set(1)</code>	<code>vs=</code>	<code>cvvis=</code>
<code>curs_set(2)</code>	<code>ve=</code>	<code>cnorm=</code>

The escape sequences for the ansi console are as follows:

```
civis= E[?25h
cvvis= E[?25hE[34lE[52;57w
cnorm= E[?25l
```

Q: I am having problems linking the correct libraries into my `curses` program. Why can't the linker find the `curses` library on the system?

A: Depending on the exact compiler flags used (`-M2l`, `-M3s`, `-xenix`, ...) and exactly which Development System is being used (80286, 80386, XENIX, UNIX), the name of the library is one of the following:

<code>/lib/Slibcurses.a</code>	small model library
<code>/lib/Mlibcurses.a</code>	medium model library
<code>/lib/Clibcurses.a</code>	compact model library
<code>/lib/Llibcurses.a</code>	large model library
<code>/lib/386/Slibcurses.a</code>	Object Module Format (OMF) XENIX small model library 80386
<code>/usr/lib/libcurses.a</code>	Common Object File Format (COFF) UNIX system library

NOTE Some of these libraries may not exist on your system. For example, the UNIX library is not found on a XENIX development system.

You may also be getting unresolved externals resulting from using the wrong libraries when linking the program.

In all SCO Development Systems starting with the SCO XENIX Development System Release 2.2, the user is prompted during installation to choose one of the following default **curses** packages:

- terminfo
- termcap
- no default package

If you choose terminfo, the **curses** libraries (*libcurses.a*) are created by linking them to the corresponding terminfo libraries (**tinfo**) and the file */usr/include/curses.h* is modified such that it defaults to terminfo **curses**.

If you choose termcap, the **curses** libraries (*libcurses.a*) are created by linking them to the corresponding termcap libraries (**tcap**) and the file */usr/include/curses.h* is modified such that it defaults to termcap **curses**.

If you choose no default package, no **curses** libraries are created and the file */usr/include/curses.h* does not default to termcap or terminfo **curses**. You, the developer, are forced to define the particular type of **curses** desired. See the “Screen Processing” chapter in your *C Language Guide*. If you wish to change your system default to terminfo or termcap, you should remove and reinstall the Development System so that these files are created correctly. To link the correct termcap **curses** libraries, use the following command:

```
cc file.c -ltermcap -ltcap
```

To link the correct terminfo **curses** libraries, use the following command:

```
cc file.c -ltinfo -lx
```

NOTE As stated previously, **-ltinfo** or **-ltcap** may be replaced by **-lcurses** if your system is set up to default to the correct **curses** environment and you are including the file *curses.h* in your C program.

Q: Why do I get fatal errors when I include *term.h* in my program.

A: Symbols are undefined if the *term.h* file is included by itself. You must include */usr/include/tinfo.h* before including */usr/include/term.h*.

Q: I am using terminfo **curses**. The manual indicates that `curs_set()` returns **ERR** if the terminal does not support the various visibility options for the cursor. How can I determine if my terminal supports a capability?

A: Support of the desired **curses** function is determined by whether or not the terminfo description for your current terminal type includes a control/escape string that performs the desired function. `curs_set()` returns **ERR** under the following conditions:

- If "attr" is < 0 or > 2.
- If "attr" is a capability that is NOT in the terminfo description; for example, invisible = "civis", normal = "cnorm", visible = "cvvis".

Confirm that the terminfo description for your terminal supports the functions needed.

Q: Why do I get the following error message when I compile a program with the `getch` system call?

```
undefined      first reference
symbol         in file
getch          myprogram.o
ld fatal: symbol referencing errors. No output written to myprogram
```

A: Although the libraries `/lib/Slibc.a`, `/lib/Mlibc.a`, `/lib/Clibc.a`, and `/lib/Llibc.a` all have references to the `getch` function, the actual resolution for the system call is in the **curses** library. If the Development System was installed with a *default curses* library, this is called `/lib/?libcurses.a`, where ? is replaced with one of the letters S, M, C, L. To resolve the system call, you must include the **curses** library. For example:

```
cc myprogram.c -lcurses
```

NOTE For `getch` to be used as a *hot key*, you must use the termio `ioctl` system call and turn off canonical input processing.

Q: Why do I get the following error message when running a program compiled with the termcap or terminfo **curses** libraries?

```
"Sorry, I don't know anything about your _____ terminal"
```

A: The termcap or terminfo description for the terminal is not installed on the machine, or the **TERMCAP** or **TERMINFO** system variables are not set. Make sure the **TERMINF** package is installed under the SCO XENIX Operating System. Make sure there exists a termcap description in the file `/etc/termcap`. **TERMINFO** descriptions are kept in the file `/usr/lib/terminfo/terminfo.src`. Make sure the **TERMCAP** or **TERMINFO**

system variables are set in your environment with the following commands in your *.profile*:

```
TERM=ansi
TERMINFO=/usr/lib/terminfo
TERMCAP=/etc/termcap
export TERM TERMINFO TERMCAP
```

In your *.login*, it is:

```
TERM=ansi
TERMINFO=/usr/lib/terminfo
TERMCAP=/etc/termcap
```

The command **env** can be used to verify that these variables are set.

Q: I am developing a program under the SCO UNIX System V/386 or SCO Open Desktop Development System that is intended to run under SCO XENIX 286 Development System. If I use the **curses** function, **printw()**, my program hangs. The same program works fine when compiled with the SCO XENIX Development System. Why?

A: The following source code can be used in your program. It is equivalent to the **printw()** function in the library, but compiles and works correctly.

```
#include <curses.h>
#include <varargs.h>

int printw(va_alist)
va_dcl
{
    register unsigned char *fmt;
    va_list ap;
    unsigned char buf[BUFSIZ];
    int n;

    va_start(ap);
    fmt=va_arg(ap, unsigned char *);
    n=vsprintf(buf,fmt,ap);
    va_end(ap);
    if (n==ERR) return ERR;
    return waddstr(stdscr,buf);
}
```

Q: Why do I get the following error message when I attempt to compile a **curses** program using the **rcc** compiler?

```
/usr/include/curses.h: 51: undefined control
```

A: The **rcc** compiler does not support use of the **#error** preprocessor directive and, therefore, cannot complete when it encounters that preprocessor directive in the file */usr/include/curses.h*. If it is necessary to use the **rcc** compiler for a **curses** compilation, the offending preprocessor directives

must be removed from the file */usr/include/curses.h*. The lines in question follow:

```
#if defined( M_TERMINFO ) && defined( M_TERMCAP )
#error Only one of "M_TERMCAP" or "M_TERMINFO" should be #define'd.
#endif
```

It is necessary to either comment out or remove these lines from the header file to complete compilation.

NOTE These preprocessor lines test to make sure that **M_TERMINFO** and **M_TERMCAP** are not both defined in your source file. If you remove these lines, you must make sure that both are not defined in your **curses** source.

NOTE The **cc(CP)** compiler does support the **#error** preprocessor directive and will, therefore, have no problem interpreting the *curses.h* file. It is recommended that **cc(CP)** be used for **curses** source compilation if at all possible.

Q: How do I change the terminal type in the middle of a **curses** program?

A: The **curses** function **newterm()** can be used to change the terminal type. Instead of calling **initscr()**, you can call **newterm()** as the first routine, as opposed to **initscr()**, which can only be called once. **newterm()** can be called more than once from the same program. The description of **newterm()** under **curses(S)** is a bit confusing. It says that "If **newterm()** is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin()** is called." However, **endwin()** must be called before the next call to **newterm()** is made. Otherwise, the **tty** modes cannot be restored correctly.

Q: Why, when using the **curses** input functions **scanw**, **getstr**, or **getch**, must input be terminated by an explicit newline (**<Ctrl>J**)?

A: After calling **initscr()**, use the **resetty()** function to restore your terminal states. Then use **crmode()** to set **CBREAK**. Now when you use **scanw**, **getstr**, or **getch**, input is terminated by a carriage return (**<Ctrl>M**). Note that if you use **raw()**, input then must be terminated by an explicit newline. Additionally, if neither **crmode()** or **raw()** has been used, **curses** defaults to setting **raw()** when reading input.

- Q:** During installation of the SCO XENIX Development System, I am asked to choose a default terminal description library for use with the **curses(S)** screen handling package. I can choose **terminfo**, **termcap**, or no default package by selecting neither. Which **curses** package should I choose?
- A:** Until release 2.2 of the SCO XENIX Development System, **curses** was strictly **termcap curses**. Depending on which **curses** package programmers are familiar with and the source code they intend to compile, one or the other package may be more appropriate. In Berkeley UNIX environments, the **termcap-based curses** is used. In System V UNIX, **terminfo curses** is used. Programmers familiar with one will probably want to use that one. Programmers porting code from one of these environments will want to use the package suited for that environment. For people not in those categories, **termcap curses** is significantly less complex and backward-compatible with programs written on older versions of XENIX.

Creating device drivers

- Q:** I want to remove the device using interrupt 7, a parallel port, from the kernel. How can this be done?
- A:** To do this, bring the system up in single-user mode and log in as *root*.
1. Change to the directory */etc/conf/sdevice.d*. Edit the file *pa* and change the second field from a "Y" to an "N".
 2. Next, you need to modify the file */etc/conf/cf.d/mdevice*. The third field for the *pa* entry contains "ictoHr". Remove the "r" (required) from this field.
 3. Finally, relink the kernel by changing to the directory */etc/conf/cf.d* and typing:
`./link_unix`

These steps may be used as a guide for removing other devices from the kernel.

- Q:** I am writing a serial driver for a multiport board but find that I am losing interrupts. What is causing this?
- A:** The problem may lie in the interrupt routine. Interrupt routines should be designed to perform quickly. The following guidelines should be adhered to when writing interrupt routines:
- Keep the interrupt routine small (less than 50 lines of code).

- Use a higher spl, for instance spl7. A lower spl gives device drivers using higher spl's priority servicing and could cause interrupts to be dropped. spl ranges are 0-7.
- Do not make function calls from your interrupt routine.
- Do not "sleep" in an interrupt routine.

Q: I am writing a SCSI device driver and am looking for any further information about SCSI drivers. Is there any available?

A: The following documents are available from SCO Support and provide information on writing SCSI device drivers:

- *The SCO Modular SCSI Interface, Versions 0, 1, and 2.* January 1, 1991 (approximately 19 pages)
- *The SCO Device Driver Writer's Guide SCSI Supplement.* July 19, 1990 (approximately 5 pages)

Q: When compiling my device driver, I get syntax errors in *sys/param.h*.

A: You must have **M_KERNEL** defined on your compile line. Refer to the file */usr/sys/conf/makefile* for additional compile-time options.

Q: I am writing a 386 device driver but I cannot find the **in** or **out** functions.

A: These functions have been replaced. The new functions that you could use are the following:

inb input a byte from a port
inw input a word (2 bytes) from a port
ind input a double word (4 bytes) from a port
outb output a byte to a port
outw output a word (2 bytes) to a port
outd output a double word (4 bytes) to a port

These functions take the same parameters as the **in** and **out** functions.

- Q:** I'm installing my own device driver (or writing a script to install one). What major device number should I use? How do I know which ones are in use?
- A:** The program `/usr/sys/conf/majorsinuse` lists all major device numbers currently in use. You should use the next sequential number for your device. For versions of SCO XENIX between 2.2.0 and 2.2.3, the following Bourne shell fragment can be used to determine the next free major number. After version 2.2.3 it is possible to use the utility `/usr/sys/conf/nextmajor` to get an available major number.

```
:
# Store list of currently used major numbers.
majors='/usr/sys/conf/majorsinuse'
# Initialize nextmajor
nextmajor=

# getmajor goes through list of currently used major numbers until
# it finds a number not in that list, this unused major number is
# stored in the variable next major.
getmajor()
{
    for majornum in $majors
    do
        if [ $nextmajor -ne $majornum ]
        then
            #echo "Major number $nextmajor is not in use."
            return
        else
            nextmajor='expr $nextmajor + 1'
        fi
    done
}
```

Files

- Q:** How can I open as many files as possible in my DOS program?
- A:** In a DOS binary compiled with the SCO UNIX System V/386 Development System, you can open up a maximum of 20 files at a time. However, there are five files that are always opened by default under DOS for each running program. These files are: *stdin*, *stdout*, *stderr*, *stdaux*, and *stdprn*. Unless these files are closed, you can only open 15 files. The following program demonstrates how to close these files.

NOTE Microsoft provides extended libraries with its C compiler that allows you to open more than 20 files at a time. Unfortunately, these libraries are NOT available for the SCO UNIX System V/386 Development System.

```

/* Program demonstrates how to open the maximum number of files when creating
   a DOS binary.

   Compile the program with the -dos flag, for example:

       cc -dos file.c
*/
#include <stdio.h>
#define MAX_FILES 60

main()
(
    int i;
    char name[6];
    FILE *fpointer[MAX_FILES];

    /* close stdin, stderr, stderr, stderr so we can open up more files */
    fclose(stdin);
    fclose(stderr);
    /* fclose(stdout); */ /* Leave stdout open so we can print to the screen */
    fclose(stderr);
    fclose(stderr);

    strcpy (name, "file0");
    fpointer[0] = fopen(name, "w");
    printf ("Opening %s ...0, name);

    /* Keep opening files until fopen fails */
    for (i = 0; fpointer[i] != NULL && i < MAX_FILES - 1; ++i) {
        sprintf (name, "file%d", i + 1);
        fpointer[i + 1] = fopen(name, "w");
        printf ("Opening %s ...0, name);
    }

    printf ("Number of files opened = %d0, i);

    /* Close all the files that we just opened */
    for ( ; i != 0; --i) {
        fclose(fpointer[i - 1]);
        printf ("Closing file%d ...0, i - 1);
    }
}

```

Q: How would I increase the number of files that one process can open?

A: Three file descriptors are inherited by every process: standard input, standard output, and standard error. If these descriptors are closed, a process can open more files. The numbers associated with these descriptors are 0, 1, and 2, respectively.

Q: Does the Per Process File Table size limit how many files (including standard descriptors) can be opened by a process?

A: The table's size can be modified. There is a kernel tunable parameter called **NOFILE** that sets the Per Process File Table size. **NOFILE** is set by default to 60, but it can range from 20 to 100.

Q: When a process exits, what happens to its active file descriptors?

A: The kernel examines the file descriptor table for the process and internally closes each open descriptor. This ensures no file remain open after the process exits. In addition, if the file descriptor corresponds to a device special file and it's the only descriptor with that device file open, the **xx_close** routine for the related driver does get called.

SCO recommends that developers still call **close(S)** explicitly for each file opened.

Cross development

Q: Why does the **xlist(S)** system not resolve when porting a program from the SCO XENIX Development System to the SCO UNIX System V/386 Development System.

A: Use the **nlist(S)** system call in place of **xlist(S)** for the SCO UNIX System V/386 Development System Release 3.2.

Q: As a DOS developer who wants to port my programs to SCO XENIX, what issues should I be aware of?

A: If you write nice, clean C, don't access the hardware directly, and use only standard I/O libraries, you should have a fairly easy time. In general, the following are issues of concern:

- Terminal independence. DOS applications typically only worry about the console, and thus need work to use **termcap** and run on other types of terminals. Lattice does have a **curses** library that would make conversion to SCO XENIX **curses** easier.

- Self-modifying code. Programs that change (or patch) themselves to work under SCO XENIX.
- Overlays also do not work with SCO XENIX.
- Program size. More than 64 kbytes of data is a problem on 8086-based processors. You can accomplish this with shared data, even on an 8086, but you must use far pointers. It is possible on a 286 to run bigger programs on SCO XENIX than on DOS, because DOS has a 640 kbytes total address space limit.
- Assembly language routines. These are dangerous, especially on 8086, particularly if they involve `cs`, `ds`, `es`, or `ss`. These routines are also risky (less so) on 286, particularly if they involve `cs` and `ss`.
- Common DOS tricks, like polling the keyboard in a loop, can cause performance and implementation problems under SCO XENIX.
- File-naming conventions are different between DOS and SCO XENIX, so applications with built-in filenames need to be changed.

Q: Why do I get the following error message when I run `adb(CP)` on a 286 binary:

```
"adb: can't read 286 emulator symbol table"
```

A: The file permissions for `/bin/x286emul` must be readable by all. To correct the file permissions, perform the following steps:

1. Login as `root`.
2. Type:

```
chmod 755 /bin/x286emul
```

Q: When using the linker, the error model mismatch or segment defined both 16- and 32-bit appears. This can occur when linking modules of a C program, or installing a device driver into the 386 kernel.

A: One of the modules of the C program or the device driver is a 286 object module, which cannot be linked into a 386 program or the 386 kernel. If the module is part of a C program, all modules must be compiled for the same processor type, either 286 or 386. If the module is a device driver, obtain a 386 version of the driver from the vendor.

Q: Why can I compile a 286 binary without error but when I try to compile it with the SCO XENIX 386 C compiler to create a 386 binary, the same code generates assertion or redefinition errors?

A: The source code makes use of the **far**, **huge**, or **near** keyword. This keyword was understood by the 286 compiler to indicate that the variable was meant to address a **far** 286 segment (32-bit address). The 386 compiler interprets this keyword to indicate that this variable was meant to address a **far** 386 segment (48-bit address). Later passes of the compiler do not handle this form of address.

Add the flag **-Dfar=** to your **cc** command line when compiling the program as a 386 binary. This causes the preprocessor to replace all occurrences of "far" with "". For example, if your 286 compile command was:

```
cc -M2me shm.c -o shm
```

You should use the following command line instead:

```
cc -M3e -Dfar= shm.c -o shm
```

Q: Why doesn't **bsearch(S)** correctly search a table in a SCO XENIX (OMF) binary created with the SCO UNIX System V/386 Development System?

A: The *bsearch.o* file needs to be pulled from the appropriate SCO XENIX Development System library and added to the SCO UNIX System V/386 Development System library. Perform the following steps:

1. Extract the file from the SCO XENIX Development System Release 2.3.1:

```
cd /lib/386
ar xv Slibc.a bsearch.o
```

2. Copy it to your SCO UNIX System V/386 machine and put it in the */lib/386* directory.

3. On the SCO UNIX System V/386 machine, replace the *bsearch.o* file in *Slibc.a* with the new one:

```
cd /lib/386
ar rv Slibc.a bsearch.o
```

4. Because this is an OMF library, **ranlib** has to be run on it again:

```
ranlib Slibc.a
```

5. You can now create the OMF binary successfully:

```
cc -x2.3 your_file.c
```

Using make

Q: Why do I get the following error when I type **make** to run my makefile?

"Make: Must be a separator on rules line 3. Stop."

A: The first character on each "action" line must be the tab character. For example:

```
#Sample makefile
target:dependents
<TAB>actions
```

Here are two sample makefiles; one that fails and one that works.

```
#Sample makefile that fails with the error:
# Make: Must be a separator on rules line 4. Stop.
foo: foo.c
cc foo.c -o foo #This line is missing the <TAB> character.

#Sample makefile that works correctly
foo: foo.c
cc foo.c -o foo #This line starts with the <TAB> character.
```

Q: When I run my makefile, I get errors from **grep(C)**, **stty(C)**, or some other system utility, even though I am not using them within my makefile, and **make** terminates. Why?

A: Try adding **SHELL=/bin/sh** to the very first line of your makefile. If this solves the problem, then there is a command in your **.cshrc** file that is causing **make** to fail. **make** sources your **.cshrc** file for each command (action) it executes. Adding the line **SHELL=/bin/sh** corrects this problem.

Q: I have neglected to include libraries or update the existing ones. When I run my makefile, the compile fails with unresolved externals and my existing working executable is overwritten with a corrupt executable. Is there a way to prevent this from occurring?

A: This is standard behavior for SCO XENIX and SCO UNIX compilers. To prevent this from occurring, if you are using **cc** from the command line, always save existing working executables to another directory or filename. If compiling with a makefile, a small modification prevents the existing executable from being overwritten. An example follows:

```
test2: test2.o
cc test2.o -o tmp_test2
mv tmp_test2 test2
test2.o: test2.c
cc -c test2.c
```

Q: I have source files spread over a number of directories. How do I write a makefile to create the desired target successfully? How do I write a recursive makefile?

A: The following example uses this directory structure: */src* contains the file *main.c*, and the subdirectory */greetings* contains *goodbye.c* and *hello.c*. The file *main.c* calls routines in *goodbye.c* and *hello.c*.

```

/src
|
-----
| |
/greetings      main.c
|
-----
| |
goodbye.c       hello.c

```

The makefile presented makes use of internal macros for convenience and flexibility. Three internally generated macros are:

\$@ full target name of the current target

\$? set to the string of names that were found to be younger than the target

\$< name of the related file that caused the action

A total of six other macros exist: **\$(@D)**, **\$(?D)**, **\$(<D)**, **\$(@F)**, **\$(?F)**, **\$(<F)**. The D refers to the directory and F to the filename. Example: If the target is */src/greetings/hello.o*, then

```

$(@D) = "/src/greetings"
$(@F) = "hello.o"

```

The final makefile appears below. Several aspects deserve further discussion:

- Notice how it uses internal rules to make the object files *hello.o* and *goodbye.o* in the makefile */src/greetings/makefile*.
- If either of these files is out of date, we change directory to *greetings* and make that target.

```
cd $(<D); $(MAKE) $(@F)
```

- The **cd** command and **make** must be done on the same line because each action line is passed to a separate invocation of the shell.

The file `/src/makefile` looks like this:

```
main: main.o greetings/hello.o greetings/goodbye.o
cc -o main main.o greetings/hello.o greetings/goodbye.o
greetings/hello.o: greetings/hello.c
cd $(<D); $(MAKE) $(@F) # if necessary, just make hello.o
greetings/goodbye.o: greetings/goodbye.c
cd $(<D); $(MAKE) $(@F) # if necessary, just make goodbye.o
```

The file `/src/greetings/makefile` looks like this:

```
hello.o:
goodbye.o:
```

Q: How do I add new `.SUFFIXES` rules in my makefile?

A: A common problem, when using `.SUFFIXES`, is neglecting to have a space character between each of the suffixes in the list. For example, the `.SUFFIXES` below do not work because `.frm.inp` is interpreted as a single suffix instead of two suffixes. This can be seen if you execute **make -p** to print out the complete set of macro definitions.

```
.SUFFIXES .frm.inp
```

The suffixes list must have white space between each suffix as in:

```
.SUFFIXES .frm .inp
```

Q: Can I write a double suffix rule where the target and dependent file aren't in the same directory?

A: No, you can't. A double suffix rule assumes that the target and dependent are in the same directory. This does not necessarily have to be the current directory. For example, the internal `.c.o` rule assumes both the object file and the source file are in the same directory, whatever directory the object file is in. Thus, here is a **make** dependency line of the form:

```
foobar/goober.o:
```

It is actually evaluated by the internal `.c.o` suffix rule as:

```
foobar/goober.o: foobar/goober.c
```

It is possible to write a dependency line explicitly where the target and dependents are in different directories. Here is an example:

```
foobar/goober.o: goober.c
cc -c -Fofobar/goober.o goober.c
```

However, there is no way to write a double suffix rule to create a dependency line of this form.

Q: What do I need to do to write my own internal rules for **make**?

A: There are two kinds of internal rules that **make** recognizes, double suffix and single suffix. A double suffix rule transforms a file with one suffix into a file with another suffix. For example, there is a pre-defined double suffix rule to transform a *.c* file into a *.o* file. The format for a double suffix rule is as follows:

```
.suffix1.suffix2:
command
command
...
```

Suffix 1 is the one being transformed, and suffix 2 is the one to which we are transforming, for example, *.c.o* for the pre-defined rule. You can use any strings you desire for the suffixes. However, if the suffixes are not already defined by **make**, you must add them to the **SUFFIXES** list. Here is a complete example of a rule to transform a *.list* file into a *.sort* file. It is assumed that *.list* files contain lists of numbers that are in no particular order, and *.sort* files contain numerically ordered lists of these numbers. The following makefile creates the *ages.sort* file from an appropriate *ages.list* file.

```
.list.sort:
sort -o$@ -n $<
.SUFFIXES: .list .sort
ages.sort:
```

The other kind of rule is the single suffix or null suffix rule. It transforms a file with a suffix to a file without any suffix at all. For example, there is a pre-defined single suffix rule to transform a *.c* file, call it *test.c*, to the file *test*.

Single suffix rules are very similar to double suffix rules except for the fact that the rule only contains one suffix. Again, that suffix must be added to **SUFFIXES** if it has not already been defined.

Here is an example of a single suffix rule to transform a *.Z* file to a file without a suffix. It is assumed that the *.Z* file was created as the result of the **compress(C)** command, and the file without the extension is uncompressed. The following example uncompresses the file *goober.Z*.

```
.Z:
uncompress $<
.SUFFIXES: .Z
goober:
```

Miscellaneous

Q: Can I use the `exec` command to call a shell script from a C program?

A: Yes. Just make the appropriate call to one of the variations of `exec(S.)` The syntax is no different from when you call it to execute a binary program.

For example, the following C program calls the **batch** script that is shown below. If this C source is in the *file.c*, it can be compiled with the `cc file.c` command. To execute it, the command is:

```
./a.out a b c d
```

The script must be in a directory on your `$PATH` or the call to `execvp` fails.

file.c

```
#include <stdio.h>

main(argc, argv)
int argc;
char **argv;
{
    execvp("batch", argv);
    perror("execvp");
    exit(1);
}
```

batch shell script

```
# prints out the arguments given on the command line
set -x
for i in $*
do
    echo $i
done
```



Q: How do I determine which terminal device is associated with my process?

A: Use the `ttname(S)` function to find the name of the terminal device associated with a file descriptor. Every process has three descriptors open by default: *stdin*, *stdout*, and *stderr*. These descriptors are associated with the terminal that the process is running on unless they have been changed. The following program shows how to use `ttname`.

```
#define STDIN 0

main()
{
    if (!isatty(STDIN))
        printf("error: stdin not associated with a terminal\n");
    else
        printf("terminal is %s\n", ttname(STDIN));
}
```

Q: How do I find out what colors a monitor is currently using?

A: Use the `ioctl(S)` parameter, `GIO_ATTR`, which returns an `int` that stores the foreground color in its two low-order bytes, and the background color in its two high-order bytes. This is demonstrated in the following program:

```

1 #include <sys/console.h>

2 main()
3 {
4     int cur_color;
5     int foreground, background;

6     cur_color = ioctl(0, GIO_ATTR, (char*)0);

7     if (cur_color == -1)
8         cur_color = 0x0007;

9     system("setcolor red white");
10    printf("This is the test");

11    foreground = cur_color & 0x0F;
12    background = (cur_color >> 4) & 0x0F;
13
14    printf(" 33[=d%c", foreground, 'F');
15    printf(" 33[=d%c", background, 'G');
16    printf(" 33[0m");
17    printf("\n\n");

18    sleep(1);
19    system("clear");
20 }
```

Line 5 Get the current colors.

Lines 6-9 If there is a problem, set current colors to black and white.

Lines 10-11 The foreground color is stored in the two low-order bytes of the `int`, and the background color is stored in the two high-order bytes of the `int`.

Lines 12-15 See the *screen*(HW) manual page for an explanation of the values used here.

For more information, see the *screen*(HW) and *setcolor*(C) manual pages.

Please help us to write computer manuals that meet your needs by completing this form. Please post the completed form to the Technical Publications Research Coordinator nearest you: The Santa Cruz Operation, Ltd., Croxley Centre, Hatters Lane, Watford WD1 8YN, United Kingdom; The Santa Cruz Operation, Inc., 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061, USA or SCO Canada, Inc., 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

Volume title: _____
(Copy this from the title page of the manual, for example, SCO UNIX Operating System User's Guide)

Product: _____
(for example, SCO UNIX System V Release 3.2 Operating System Version 4.0)

How long have you used this product?

- ☐ Less than one month ☐ Less than six months ☐ Less than one year
☐ 1 to 2 years ☐ More than 2 years

How much have you read of this manual?

- ☐ Entire manual ☐ Specific chapters ☐ Used only for reference

	Agree			Disagree		
The software was fully and accurately described	☐	☐	☐	☐	☐	☐
The manual was well organized	☐	☐	☐	☐	☐	☐
The writing was at an appropriate technical level (neither too complicated nor too simple)	☐	☐	☐	☐	☐	☐
It was easy to find the information I was looking for	☐	☐	☐	☐	☐	☐
Examples were clear and easy to follow	☐	☐	☐	☐	☐	☐
Illustrations added to my understanding of the software	☐	☐	☐	☐	☐	☐
I liked the page design of the manual	☐	☐	☐	☐	☐	☐

If you have specific comments or if you have found specific inaccuracies, please report these on the back of this form or on a separate sheet of paper. In the case of inaccuracies, please list the relevant page number.

May we contact you further about how to improve SCO UNIX documentation?
If so, please supply the following details:

Name _____ Position _____

Company _____

Address _____

City & Post/Zip Code _____

Country _____

Telephone _____ Facsimile _____



10 December 1991



BH01 409P000

58085



AT01400P000