ALLEN-BRADLEY

OS-9 C Language

User Manual

Acknowledgements

The OS-9/68000 Source Level Debugger was written by Richard Russell. Special thanks are given to Larry Crane, Kim Kempf and Robert Doggett for their infinite patience and their invaluable design and implementation ideas. Thanks also to James Jones, Tim Harris, Todd Earles, Tony Hoffman and Dave Lyons for their use and testing of the software.

Copyright and Revision History

OS-9 C Compiler and OS-9 C Functions

Copyright © 1991 Microware Systems Corporation. All Rights Reserved. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from Microware Systems Corporation.

This manual reflect Version 3.1 of the C Compiler. Version 3.1 of the C Compiler is to be used with Version 2.3 or greater of the OS-9 Operating System.

Publication Editor: Walden Miller, Eileen Beck
Revision: I
Publication date: March 1991
Product Number: CCC-68-NA-68-MO

OS-9/68000 Source Level Debugger

Copyright © 1987 Microware Systems Corporation. All Rights Reserved. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual or otherwise is prohibited, without written permission from Microware Systems Corporation.

This manual reflects Version 2.0 of SrcDbg. Version 2.0 of Srcdbg is to be used with Version 2.2 or greater of the OS-9/68000 Operating System, Version 3.0 or greater of the C Compiler, Edition 53 or greater of L68, Edition 54 or greater of R68, Edition 77 or greater of R68020.

Publication Editor: Walden Miller, Kathleen Flood
Revision: B
Publication date: July 1988
Product Number: SDG-68NA-68-MO

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, Microware will not be liable for any damages, including indirect or consequential, from use of the OS9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction Notice

The software described in this document is intended to be used on a single computer system. Microware expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of Microware and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

For additional copies of this software and/or documentation, or if you have questions concerning the above notice, the documentation and/or software, please contact your OS-9 supplier.

Trademarks

Microware and OS-9 are registered trademarks of Microware Systems Corporation and OS-9/68000 is a trademark of Microware Systems Corp.
UNIX is a trademark of Bell Laboratories.
VMS is the trademark of Digital Equipment Corp.
SUN is the trademark of Sun Microsystems, Inc.

Microware Systems Corporation w 1900 N.W. 114th Street
Des Moines, Iowa 50325-7077 w Phone: 515/224-1929

Introduction

Microware's C Language Compiler System is an advanced technology, high-performance software development tool with the following features:

- comprehensive implementation of the full language
- extremely efficient code generation producing extremely fast and compact object programs
- generates position-independent, reentrant, ROMable code
- high compilation speed
- UNIX and OS-9® compatible standard libraries

68000-family microprocessors, the OS-9 operating system and the C language form an outstanding combination. The 680x0's stack-oriented instruction set and versatile repertoire of addressing modes handle the C language very well. The OS-9 C compiler was designed specifically for the 68000 family and takes full advantage of their abilities and features.

This compiler also serves as a gateway between UNIX and OS-9. Because of the compiler compatibility and similarities of OS-9 and UNIX, almost any application program written in C can be transported from a UNIX system to an OS-9 system, recompiled and correctly executed. The compiler can also be run on UNIX-based computers and the output downloaded to an OS-9-based 680x0 system.

The 68020 C Compiler

The 68020 C Compiler employs the additional capabilities of the 68020 MPU and the 68881 math coprocessor.

This manual covers both the 68000 and 68020 C Compiler. The 68020 Compiler can process all 68000 code and syntax; however, there are additional libraries available to the 68020 Compiler. Because of this discrepancy, all items that are specific to the 68020 compiler are shown in shaded boxes for easy reference. All other text references both the 68000 and 68020 compilers. All references to OS-9/68000 or 68000 code includes 68020, unless specifically disclaimed.

Cross Compiler Versions

The OS-9 C compiler is available in cross compiler versions for use on various development systems. The systems currently supported are listed below:

Host system:	Operating system:
DEC VAX Series	BSD 4.2, BSD4.3,VAX/VMS
SUN/3 Series	SUN UNIX

Unless otherwise noted, the information given in this document applies to native mode and cross versions of the compiler.

The C Programming Language - Kernighan & Ritchie

The specification for the OS-9 C compiler is the book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Inc. It is hereafter referred to as “K&R”. This compiler conforms exactly to this specification except for the implementation dependent characteristics noted in this manual.

K&R frequently refer to characteristics of the C language whose exact operations depend on the architecture and instruction set of the computer actually used. This manual contains specific information regarding this version of C for 68000 family processors.

SRCDBG 2.0 Caveats

The following caveats should be noted when using Srcdbg. Most of these problems will not arise with normal usage of Srcdbg. However, these explanations are provided for the user in case such situations do arise.

Using Cstart

Register .a5 is used as the frame pointer by the C Compiler. Because of this, Srcdbg assumes that register .a5 is set to zero before reaching the first link instruction generated by c68. This is done automatically by “cstart”. If cstart is not used (i.e. the user has written a specific routine to take its place), .a5 must be set to zero in order for Srcdbg to work correctly.

Programs Which Chain

Srcdbg is not able to continue debugging a program which chains to other programs or chains to itself after the chain occurs. This is because of the necessity of having a correct symbol table for the program being debugged. If a program chains to another program, there will be no symbols available to debug the new program. If a program chains to itself, the symbol table will be there, but the addresses of the code and data areas usually will have moved. Consequently, the symbol table is unusable.

This will be corrected in a future release.

Bus Errors

If a bus error occurs on a system with a non-68020 processor, Srcdbg may not correctly stop at the source line at which the error occurred. It may be off by a line or two. This is due to the fact that the bus error PC register may be off from zero to ten bytes for any non-68020 processor. Srcdbg will accurately reflect bus errors occurring on 68020 systems.

Tagged Structs, Unions and Enums

Srcdbg issues a warning if a tagged struct, union or enum of one file has the same tag name as a different tagged struct, union or enum of another file. For example:

ATTENTION: “<s_u_e1>” in “<file1>” differs from “<s_u_e2>” in “<file2>”

Important: <s_u_e> indicates the struct, union or enum tag name.

Using Files with the Same Name

Srcdbg issues a warning if a source or include file with executable code is referenced more than once. For example:

ATTENTION: file “<filename>” with executable code has been previously seen. The name for this file is now “<filename><num>”.

Multiple files will be named <filename><num>, where <num> starts at zero. For example, the second file named program.c will be referred to as program.c0 by SrcDbg.

Edited Source files

Srcdbg issues a warning if a sourcefile has been edited since it was compiled with the –g option. For example:

ATTENTION: source file “<filename>” has been changed.

If the source code has been changed, the file needs to be recompiled with the –g option to ensure that the symbol information matches the source code.

Important: If a symbol file is used that does not actually match the program module, SrcDbg’s behavior will be unpredictable.

SECTION 1 OS-9 C COMPILER

Installing and Running the Compiler

Chapter 1

Executable Files 1-1
Library Files 1-2
Definition Files 1-3
Library and Definition File Directory Searching 1-3
Command Lines and the C Executive 1-4
Compiler Options 1-7
Example Command Lines 1-9
Math Library and C Library Selection Options 1-9
Notes 1-10

Compiler Implementation

Chapter 2

Data Representation 2-1
Register Variables 2-1
Access To Command Line Parameters (Argc, Argv, Envp) 2-3
End-of-Line Character 2-3
Implementation-Dependent Variances 2-3
Enhancements and Extensions 2-3
System Calls and the Standard Library 2-7

Compiler Organization

Chapter 3

Compiler Organization 3-1

Using the Termcap Library

Chapter 4

Using the Termcap Library 4-1

SECTION 2 THE C STANDARD LIBRARY

The C Standard Library

Chapter 5

The C Standard Library 5-1
_atou() 5-7
_cmpnam() 5-7
_cpymem() 5-8
_errmsg() 5-9
_ev_creat() 5-10
_ev_delete() 5-11
_ev_info() 5-12
_ev_link() 5-12
_ev_pulse() 5-13
_ev_read() 5-14
_ev_set() 5-14
_ev_setr() 5-15
_ev_signal() 5-16
_ev_unlink() 5-17
_ev_wait() 5-18
_ev_waitr() 5-19
_exit() 5-20
_freemin() 5-21
_get_module_dir() 5-22
_get_process_desc() 5-23
_get_process_table() 5-24
_get_sys() 5-24
_gs_devn() 5-25
_gs_eof() 5-26
_gs_gfd() 5-27
_gs_opt() 5-28
_gs_pos() 5-29
_gs_rdy() 5-30
_gs_size() 5-31
_julian() 5-32
_lcalloc() 5-34
_lfree() 5-35
_lmalloc() 5-36
_lrealloc() 5-37
_mallocmin() 5-38
_mkdata_module() 5-39
_parsepath() 5-40
_prgname() 5-40
_prsnam() 5-41
_setcrc() 5-41

The C Standard Library

Chapter 5 Continued

_setsys() 5-42
_srqmem() 5-43
_srtmem() 5-44
_ss_attr() 5-45
_ss_dcoff() 5-46
_ss_dcon() 5-46
_ss_dsrts() 5-47
_ss_enrts() 5-47
_ss_lock() 5-48
_ss_opt() 5-49
_ss_pfd() 5-50
_ss_rel() 5-51
_ss_rest() 5-52
_ss_size() 5-52
_ss_ssig() 5-53
_ss_tiks() 5-54
_ss_wtrk() 5-55
_strass() 5-56
_sysdate() 5-57
_sysdbg() 5-59
_tolower() 5-59
_toupper() 5-60
abs() 5-60
access() 5-61
acos() 5-61
alm_atdate() 5-62
alm_atjul() 5-63
alm_cycle() 5-64
alm_delete() 5-65
alm_set() 5-65
asctime() 5-66
asin() 5-67
atan() 5-67
atof() 5-68
atoi() 5-68
atol() 5-69
attach() 5-69
calloc() 5-70
ceil() 5-71
chain(), chainc() 5-72
chdir() 5-74
chmod() 5-75
chown() 5-76
chxdir() 5-77

The C Standard Library

Chapter 5 Continued

cleareof() 5-78
clearerr() 5-79
clock() 5-80
close() 5-81
closedir() 5-82
cos() 5-82
crc() 5-83
creat() 5-85
create() 5-86
ctime() 5-87
detach 5-88
difftime() 5-89
dup() 5-89
ebrk() 5-90
exit() 5-91
exp() 5-91
fabs() 5-92
fclose() 5-92
fdopen() 5-93
feof() 5-94
ferror() 5-95
fflush() 5-96
fgetc() 5-97
fgets() 5-97
fileno() 5-98
findnstr() 5-98
findstr() 5-99
floor() 5-99
fopen() 5-100
fprint() 5-102
fputs() 5-103
fread() 5-104
free() 5-105
freemem() 5-106
freopen() 5-107
frexp() 5-108
fscanf() 5-108
fseek() 5-109
ftell() 5-110
fwrite() 5-111
getc(), getchar() 5-112
getenv() 5-114
getime() 5-115
getpid() 5-115

The C Standard Library

Chapter 5 Continued

gets() 5-116
getstat() 5-117
getuid() 5-118
getw() 5-119
gmtime() 5-120
hypot() 5-120
ibrk() 5-121
index() 5-122
intercept() 5-123
isalnum() 5-125
isalpha() 5-125
isascii() 5-126
iscntrl() 5-126
isdigit() 5-127
islower() 5-127
isprint() 5-128
ispunct() 5-128
isspace() 5-129
isupper() 5-129
isxdigit() 5-130
kill 5-131
Idexp() 5-132
localtime() 5-132
log() 5-133
log10() 5-133
longjmp() 5-134
lseek() 5-135
makdir() 5-136
make_module() 5-137
malloc() 5-139
memchr() 5-140
memcmp() 5-140
memcpy 5-141
memmove() 5-141
memset() 5-142
mknod() 5-142
mktemp() 5-143
mktime() 5-143
modcload() 5-145
modf() 5-146
modlink() 5-146
modload() 5-147
modloadp() 5-148
munlink() 5-149

The C Standard Library

Chapter 5 Continued

munload() 5-150
open() 5-151
opendir() 5-152
os9exec() 5-153
os9fork(), os9forkc() 5-155
pause 5-156
pffinit() 5-157
pflinit() 5-157
pow() 5-158
prerr() 5-158
printf() 5-159
putc(), putchar() 5-162
puts() 5-164
putw() 5-165
qsort() 5-166
read(), readln() 5-167
readdir() 5-168
realloc() 5-168
rewind() 5-169
rewinddir() 5-169
rindex() 5-170
sbrk() 5-171
scanf() 5-172
seekdir() 5-174
setbuf() 5-175
setime() 5-176
setjmp() 5-176
setpr() 5-177
setstat() 5-178
setuid() 5-179
sigmask() 5-180
sin() 5-181
sleep() 5-182
sprintf() 5-182
sqrt() 5-183
srqcmem() 5-183
sscanf() 5-184
stacksiz() 5-185
strcat() 5-185
strcmp() 5-186
strcpy 5-186
strhcpy() 5-187
strlen() 5-187
strncat() 5-188

The C Standard Library

Chapter 5 Continued

strncmp() 5-188
strncpy() 5-189
strtod() 5-190
strtol(), strtoul() 5-191
system() 5-192
tan() 5-192
telldir() 5-193
tgetent() 5-194
tgetflag() 5-195
tgetnum() 5-195
tgetstr() 5-196
tgoto 5-197
time() 5-198
toascii() 5-199
tolower() 5-200
toupper() 5-200
tputs() 5-201
tsleep() 5-202
ungetc() 5-203
unlink() 5-204
unlinkx() 5-205
wait() 5-206
write(), writeln() 5-207

SECTION 3 OS-9/68000 SOURCE LEVEL DEBUGGER

Overview of SrcDbg

Chapter 6

Overview of SrcDbg 6-1
C Compiler Revision Requirements 6-1
Setting the Environment 6-1
The ".dbg" and ".stb" Symbol Files 6-2
Invoking SrcDbg 6-3
SrcDbg Help 6-7
Exiting SrcDbg 6-7
SrcDbg Command Syntax 6-7
SrcDbg Scope 6-7
Scope Expressions 6-10
Line Number Expressions 6-12
C Expressions 6-13
Location Expressions 6-14
Command Line Notes 6-15
Example Tutorial Program 6-15
Notes 6-18

Debugger Control Commands

Chapter 7

Fork 7-2
Go 7-4
Step 7-7
Name 7-9
Return 7-11
Break 7-13
Watch 7-16
Kill 7-19
Log 7-21
Option 7-22
Read 7-24

Data Manipulation Command

Chapter 8

List 8-2
Info 8-5
Frame 8-8
Print 8-10
Assign 8-13
Chc 8-14
Context 8-16
Find 8-17
Locals 8-18

System Commands

Chapter 9

Chd 9-2
Shell 9-3
Help 9-4
Quit 9-4
Chx 9-5
Setenv 9-6
Unsetenv 9-7

Assembly Level Commands

Chapter 10

Assembly Level Display Information 10-1
Instruction Disassembly Memory Display 10-4
Floating Point Memory Displays 10-5
Asm 10-6
Change 10-7
Dilist 10-9
Disasm 10-11
Dump 10-13
Gostop 10-15
Link 10-17
MFill 10-18
Msearch 10-20
Symbol 10-23
Trace 10-24

SrcDbg Syntax and Commands

Chapter 11

Syntax 11-1
Commands 11-1

Error Codes

Appendix A

Section 1

OS-9 C Compiler

This section of the manual describes:

- installing and running the OS-9 compiler
- implementing the OS-9 compiler
- organizing the OS-9 compiler
- using the termcap library

Chapter 1

Installing and Running the Compiler

The OS-9 C Compiler system consists of three distinct types of files: executable files, library files and definition files. This section describes the files provided with the C Compiler system, their use, and where you should install them on your system.

Executable Files

The following binary files comprise the executable part of the compiler system. The following files must be in the CMDS directory on OS-9 systems for the compiler system to work. You must copy these files from the distribution media:

Binary code:	File:
cc	Compiler executive program (cc68 for cross versions)
cpp	Macro preprocessor
c68	Compiler pass
o68	Optimizer
c68020	Compiler pass for 68020 compiler

The following additional modules are also required to run the C Compiler.

Binary code:	File:
r68	OS-9/68000 macro assembler
l68	OS-9/68000 linker
cio	Trap handler for the C I/O Library
r68020	OS-9/68020 macro assembler
cio020	Trap handler for 68020 C I/O library

Library Files

The following are important files used during the compilation process. You should locate these files in a directory named LIB on the system's default mass storage device. You must copy these files from the distribution media:

Binary code:	File:
cstart.r	Contains startup code for compiled programs.
clib.l	Contains the standard library, math functions and the system library.
clibn.l	Contains the same as clib.l but does not use the math trap handler for floating point support. The routines to perform floating point are extracted from the math.l library.
clib020.l	Contains the 68020 standard library, math functions and the system library.
clib020n.l	Contains the same as clib020.l, but does not use the math trap handler for floating point support. The routines to perform floating point are extracted from the math.l library.
clib020h.l	Contains the same as clib020n.l, but uses the 68881 coprocessor hardware instructions for floating point.

The following additional modules are also required to run the C Compiler, but are supplied elsewhere:

Binary code:	File:
math.l	Contains the same floating point functions as the math trap handler, but in stand alone form.
math881.l	Contains the same math.l, but uses 68881 hardware instructions for floating point.
sys.l	Contains the system global definitions.
termlib.l	Contains the termcap database screen manipulation functions.

Definition Files

The following are definition files that define macros, constants and data structures used by library functions. The function descriptions indicate which of these files should be included for specific functions. These files should be located in the DEFS directory on the system's default mass storage device.

Definition file:	Function description:
ctype.h	Type and macro definitions for character classification functions
dir.h	Definitions for directory manipulation functions
direct.h	Definitions of OS-9 disk directory format
errno.h	OS-9 error number definitions
events.h	OS-9 event sub-system definitions
math.h	Math function definitions
modes.h	OS-9 file permission value definitions
module.h	OS-9 memory module structure definitions
procid.h	OS-9 process descriptor structure definitions
setjmp.h	setjmp and longjmp function state buffer definitions
setsys.h	Request codes for setsys function
sgstat.h	Structure for getstat, setstat and ss_opt functions
signal.h	OS-9 signal definitions
stdio.h	Standard I/O definitions
strings.h	String handling function definitions
termcap.h	Terminal manipulation library definitions
time.h	Buffer definition for getime and setime functions
types.h	Standard type definitions

Library and Definition File Directory Searching

The C compiler system relies on a number of header (#include) and library files to compile a program. They are kept in the directories named DEFS and LIB, respectively. These directories are located on the disk device used for system file storage:

- On systems with a hard disk, the pathlists are /h0/LIB and /h0/DEFS.
- On systems without a hard disk, the pathlists are /d0/LIB and /d0/DEFS.
- On systems using a large RAM disk, the pathlists are /r0/LIB and /r0/DEFS or /dd/LIB and /dd/DEFS.

There are three distinct methods of determining the location of the library files. The following priority is used to determine the directory searched:

- command line options
- environment variables
- default search method

You can specify additional "defs" directories on the compiler executive command line. The v option specifies a directory to search for definition files (in addition to DEFS). You can use this option more than once on a command line. The directories are searched in the order provided on the command line (that is, before the system DEFS directory). Use the –w option to specify an alternative "LIB" directory. This directory is used instead of LIB. You can use this option to designate custom library files.

You can also use the shell environment variables, CLIB and CDEF, to specify the directories in which the library and definition files reside. CDEF indicates the pathlist for the directory containing the default definition files. CLIB indicates the pathlist containing the library files. Set the environment variables with the setenv utility command. These variables remain in existence for the life of the current (or future) shell.

Important: These techniques for specifying *search* directories are used mainly for cross compilers and network systems where these directories are found on a file-server node.

The C executive (cc) checks for the existence of certain devices to determine the **default device**. cc tries to access these devices in the following order: /dd, /h0, /d0 using the Microware naming conventions:

- /dd = default device (usually a RAM disk)
- /h0 = hard disk
- /d0 = floppy disk

The first device that can be accessed is assumed to contain the DEFS and LIB directories. Because all devices' directories are not searched, the directories should appear on the first device in the search list that is present on the system.

Command Lines and the C Executive

The compiler system is managed by an executive program called cc (or cc68 in cross-compiler versions). The executive accepts a command line and automatically calls the other parts of the compiler system as needed. The syntax of the command line which calls the compiler is:

```
cc [<options>] <file> {<file>} [<options>]
```

You can compile one or more files with a single command line. You may mix C or assembler source code and relocatable type files on the command line. The C executive manages the compilation through as many as four stages:

- preprocessing code
- compilation to assembler code
- assembly to relocatable module
- linking to binary executable code (in OS-9000 memory module format).

The compiler accepts three types of source files. The source files must conform to the naming convention provided below. Any of the file types may be mixed on the command line:

Suffix:	Use:
.c	C source file
.a	Assembly language source file
.r	Relocatable module

If only one source file is specified on the command line, the output file is created with a name obtained by removing the suffix from the name supplied on the command line. For example, the following command creates an executable file called prg:

```
cc prg.c
```

If multiple source files are specified on the command line, the output file is created with the name output, unless an alternative name is specified using a compiler option.

Regardless of the number of files specified on the command line, the output file is created in the current execution directory unless overridden by a compiler option. Any relocatable modules generated as intermediate files are left in the same directories as their corresponding source files with their suffixes changed to .r.

Temporary Files and Optimizing Compilation Speed

A number of temporary files are created in the current working data directory during compilation, and it is important to ensure that enough space is available on the disk drive. As a rough guide, at least three times the number of blocks in the largest source file (and its #include files) should be free.

The compilation speed is directly affected by the speed at which the temporary file can be written and read. The working directory used for compilation should be the fastest device available on the system. For example, if your OS-9 system has a memory-based virtual disk device, use it for your working directory. In addition, you can use the –t command line option to specify a fast device on which to place any temporary files.

C Library Selection Options

You can use the –i option to cause the system's installed cio trap handler module to be used for the C library rather than linking most of the C library code into the program. The cio trap handler is automatically loaded at system startup time as the standard system utilities use this facility. Systems with a 68020 MPU can use the cio020 trap handler.

Important: The module name in each of these trap handlers is cio. Simply rename cio020 to cio to load it instead of the 68000 cio module:

```
chd /h0/cmds
rename cio cio.68k
rename cio020 cio
```

When a program is linked with the cio.l library (using the cc –i option), the following functions are executed in the trap handler:

_errmsg	fdopen	getc	opendir	seekdir
_freemin	fflush	getenv	os9exec	setbuf
_prgname	fgets	gets	printf	setstat
_tidyup	fopen	getstat	putc	sprintf
access	fprintf	getw	puts	sscanf
asctime	fputs	gmtime	putw	telldir
calloc	fread	localtime	read	time
close	free	lseek	readdir	ungetc
closedir	freopen	malloc	readln	unlink
creat	fscanf	mknod	realloc	unlinkx
create	fseek	mktime	rewind	write
dup	ftell	modloadp	rewinddir	writeln
fclose	fwrite	open	scanf	

Compiler Options

The compiler recognizes many command line options which modify the compilation process. Options are not case significant. All options are recognized before compilation begins. Consequently, you can place the options anywhere on the command line. You can group options together (e.g. –sr) except where an option specifies an argument (e.g. –f=<path>).

Option:	Description:
–a	Suppresses the assembly phase, leaving the output as assembler code in a file with the .a suffix.
–bg	Sets the ***sticky*** bit in the module header to cause the module to remain in memory even if the link count becomes zero.
–bp	Prints the arguments passed to each compiler phase and an exit status message. This is useful to determine which arguments cc passes to each phase (given various other option flags).
–c	Outputs the source code as comments with the assembler code. This option is most useful with the –a option.
–d<identifier>	Is equivalent to a #define <identifier> in the source file. This option is useful where different versions of a program are maintained in one source file and differentiated through the #ifdef or #ifndef preprocessor directives. If <identifier> is used as a macro for expansion by the preprocessor, 1 (one) is the expanded value unless an expansion string is specified using the form d<identifier>=<string>.
–e=<number>	Sets the edition number constant byte to the specified number. This is an OS-9 convention for memory modules.
–f=<pathlist>	Overrides the output file naming conventions. The output file is given the name specified by the last element of <pathlist>. The module name is the same as the file name unless the –n=<name> option is used. This option causes an error if either the –a or –r option is also present. If <pathlist> is a relative pathlist, it is relative to the current **execution** directory.
–fd=<pathlist>	Is identical to the –f=<pathlist> option with the following exception: if <pathlist> is a relative pathlist, it is relative to the current ***data*** directory.
–g	Causes the linker to output a symbol module for use by the symbolic assembly language level debugger. The symbol module has the same name as the output file with .stb appended. If a STB directory exists in the target output directory, the symbol module is placed there. Otherwise, it is placed in the same directory as the output file.
–i	Links the program with the cio.l library, causing a trap handler module called cio to handle references to selected C I/O functions.
–j	Prevents linker from creating a jump table.
–k=<n>[w|l][cw|cl][f]	<n> is the target machine: 0=68000 (default), 2=68020. w causes 16-bit data offsets to be generated (default 68000). l causes 32-bit data offsets to be generated (default 68020). cw causes 16-bit code references to be generated (default 68000). cl causes 32-bit code references to be generated (default 68020). f causes c68020 to generate 68881 instructions for float/double types.
–l=<path>	Specifies a library file to be searched by the linker before the standard library, math libraries and system interface library.
–m=<mem size>	Instructs the linker to allocate <mem size> for the program stack. Memory size is given in kilobytes. The default stack size is approximately 2K.
–n=<name>	Specifies the output module's name.

Option:	Description:
–o	Inhibits the assembly code optimizer pass. The optimizer shortens object code by about 11% with a comparable increase in speed. It is recommended for production versions of debugged programs.
–q	Specifies quiet mode: the executive does not announce internal steps as they occur. Only error messages, if any, are displayed.
–r[=<dir>]	Suppresses linking library modules into executable programs. Output is left in files with a .r suffix. If –r=<dir>, then .r files stay in <dir>.
–s	Stops the generation of stack-checking code. Use this option with great care and only when the application is extremely time critical and when the use of the stack by compiler generated code is fully understood.
–t=<dir>	Causes the executive to place the temporary files used by any compiler phase in the directory named <dir>. If the device containing the directory is the ramdisk device (e.g., –t=/r0), compilation time is drastically reduced.
–u<name>	Undefines previously defined pre-processor macro names. Macro names pre-defined in the pre-processor are OSK and mc68000. These names are useful to identify the compiler under which the program is being compiled for purposes of writing machine and operating system independent programs.
–v=<dir>	Specifies an additional directory to search for pre-processor #include files. File names within quotes are assumed to be in the current directory. File names within angle brackets (< >) are searched for in the specified directory. This option may appear more than once. In this case each directory is searched in the order given on the command line. The default DEFS directory is searched after all specified directories have been searched.
–w=<dir>	Specifies the directory containing the default library files (cstart.r, clib.l, etc.). This option is useful when the library directory is on a remote file server or a custom version of such files are being used.
–x	Causes the compiler to generate trap instructions to access the floating point math routines. This option should appear on the command line when the program is both compiled and linked (if performed separately). The linker causes the object program to use the trap handler modules rather than extracting the code from the math libraries.

Example Command Lines

Command line:	Action:	Output file(s):
cc prg.c	compile to an executable program	prg (execution dir.)
cc prg.c –a	compile to assembly language source code	prg.a (current data dir.)
cc prg.c –r	compile to relocatable module	prg.r (current data dir.)
cc prg1.c prg2.c prg3.c	compile to executable program	prg1.r, prg2.r, prg3.r (current data dir.); output (execution dir.)
cc prg1.c prg2.a prg3.r	compile prg1.c, assemble prg2.a, and combine all into an executable program	prg1.r, prg.r (current data dir.); output (execution dir.)
cc prg1.c /d0/ted/prg2.c	compile to executable program	prg1.r (current data dir.) prg2.r (/d0/ted), output (execution dir.)
cc –e=3 name.c –f=prog	compile to executable program set module revision level to 3	name.r (current data dir.) prog (execution dir.)
cc sieve.c –igf=sieve	compile to executable using the cio subroutine module and make a symbol module for debug	sieve (execution dir.); sieve.stb (current data dir.)
cc prog.c –dfloats	compiles executable program with the floats definition identifier passed to the compiler.	prog (execution dir.)
cc prg.c –k2	generates code for 68020	prg (execution dir.)
cc prg.c –k2w	generates code for 68020 using 16-bit offsets	prg (execution dir.)
cc prg.c –k2f	generates code for 68020/68881	prg (execution dir.)

Math Library and C Library Selection Options

The –k and –x options for cc determine which C libraries and Math libraries are used during the compilation/linking process. The following chart shows the various configurations possible using these options:

Command:	Target:	C library:	Math library:	Description:
cc	68000	clibn.l	math.l	(Default selection) Code is generated for the 68000 MPU. The floating point math library code is linked into the program.
cc –x	68000	clib.l	<trap>	Same as cc except the system's installed math trap handler module is used instead of linking the math code into the program.
cc –k2	68020	clib020n.l	math.l	Code is generated for the 68020 MPU. The floating point math library code is linked into the program.
cc –k2 –x	68020	clib020.l	<trap>	Same as cc –k2 except the system's installed math trap handler module is used instead of linking the math code into the program.
cc –k2f	68020/881	clib020h.l	math881.l	Code is generated for the 68020 MPU. 68881 coprocessor instructions are generated inline for floating point calculations. The math881.l library contains the trig and numeric conversion functions.

Notes

Chapter 2

Compiler Implementation

Data Representation

Each variable type requires a specific amount of memory for storage. The sizes of the basic types (in bytes) are as follows:

Data type:	Bytes:	Internal representation:
char	1	two's complement binary
unsigned char	1	unsigned binary
short	2	two's complement binary
unsigned short	2	unsigned binary
int	4	two's complement binary
unsigned	4	unsigned binary
long	4	two's complement binary
float	4	binary floating point (see below)
double	8	binary floating point (see below)
"pointer to ..."	4	address

Important: The compiler follows the convention that char and short are converted to int with sign extension. The data types long and int are synonymous.

Register Variables

You can optimize code size and speed by a judicious use of register variables. When you declare heavily used variables as register type storage, the compiler can perform many optimizations on-the-fly that it could not otherwise. The most efficient use of register variables is for pointers or loop counters.

You can use register declarations on automatic variables or function arguments. The register declaration is effective only for integral or pointer types. Any invalid register types or declarations in excess of available registers are simply ignored. By declaring as many register variables as possible, the best possible object code is obtained.

Within each function, three 32-bit address (An) registers are available for pointers and four 32-bit data (Dn) registers are available for non-pointer variables. If a register declaration is inappropriate for the type (like float or double), the declaration is simply ignored (i.e., the storage class is made automatic).

The register assignments for the variables are made in the order that the declarations appear. Therefore, it is wise to declare register variables in the order of heaviest use. If more register declarations are given than registers available, the excess register storage class declarations are considered automatic.

For further details see K&R on **register declaration**.

Floating Point Format

The compiler uses a floating point representation based on IEEE Draft Standard 754, which has been adapted as the standard OS-9/68000 format. Using this format affords direct compatibility with floating point arithmetic coprocessor hardware.

Single Precision Storage Format:

Range: approximately * $1.2*10^{-38}$ to $3.4*10^{38}$

Precision: approximately 7 decimal digits

Double Precision Storage Format:

Range: approximately * $2.2*10^{-308}$ to $1.8*10^{308}$

Precision: approximately 16 decimal digits

The mantissa has an implied leading1 bit. The sign bit is the most significant bit of the value. The exponent is biased by 128 for floats and 1024 for doubles.

Floats are promoted to doubles when passed as parameters. Floating point code is normally accessed by the compiler via traps to the OS-9 floating point trap handlers, thereby saving considerable memory. The floating point math routines also promote floats to doubles for computation.

Although the compiler system uses the storage representation set forth in the specification, the actual mathematical library routines do not support certain internal details of the standard such as infinity arithmetic, NANs, user defined rounding options, etc. These functions are of use to a very limited class of users, do not affect future compatibility, and if supported (in software routines) would significantly degrade execution speed.

Access To Command Line Parameters (Argc, Argv, Envp)

The standard C arguments, argc and argv, are available to main as described in K&R. The cstart.r start-up code converts the parameter string passed to it by the parent process into null-terminated strings as expected by the K&R standard. The argument envp points to a list of environment variables for the process. The environment variables are normally set by the shell. See the discussion of the cstart.r startup module in Chapter 3 for the details on argument and environment variable handling.

End-of-Line Character

The escape sequence for new-line (\n) refers to the ASCII carriage return character (used by OS-9 for end-of-line), not linefeed (hex 0A) as used in UNIX. Despite this difference, programs using \n for end-of-line (including all programs in K&R) work properly.

Implementation-Dependent Variances

Although C is a very portable language, there are inevitably minor differences between versions. This compiler is no exception, and its differences mostly reflect parts of C that are either obsolete or hardware dependent.

Enhancements and Extensions

The compiler includes a number of additional useful features as listed below. These features are non-standard and may affect the portability of programs to other C compilers.

Embedded Assembly Language

As versatile as C is, occasionally there are some things that can only be done (or done at maximum speed) in assembly language. The OS-9 C compiler permits user-supplied assembly language statements to be directly embedded in C source programs.

A line beginning with #asm switches the compiler into a mode which passes all subsequent lines directly to the assembly language output until a line beginning with #endasm is encountered. #endasm switches the mode back to normal.

Alternatively, a single line beginning with an at sign (@) causes the compiler to pass the remainder of the line directly to the assembler.

You should exercise care when using embedded assembly language so that the correct code section is adhered to. Normal code from the compiler is in the psect (code) section. If your assembly code uses the vsect (variable) section, be sure to put an endsect directive at the end to leave the state correct for the following compiler generated code.

The –a and –c compiler options are useful to check out the embedded assembler code.

Consult the section of this manual entitled **Compiler Organization** for complete details on assembly language C functions.

The Remote Storage Class

The C compiler supports an additional storage class: the remote storage class.

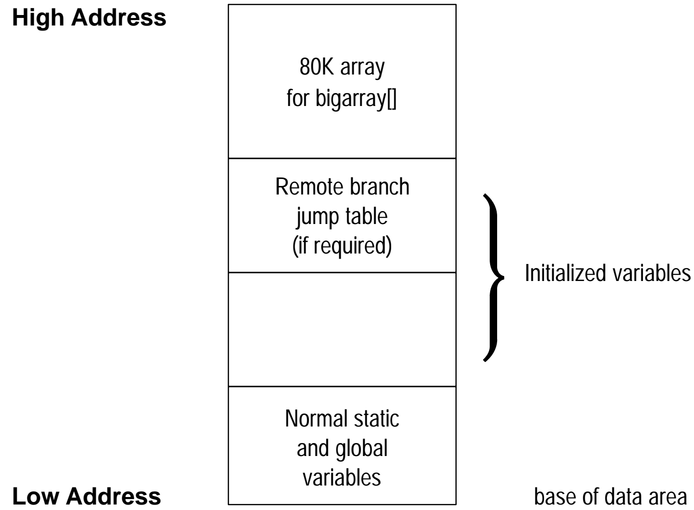
The 68000 register indirect with offset addressing mode limits the addressing range to 64K. If a program requires more than 64K of static or global storage, you must use the **indexed register indirect with offset** addressing mode. This is the only way to index an address register with a 32 bit offset.

The compiler generates variable references with 32 bit offsets for those variables declared as remote. All remote data for a program is assigned by the linker after the normal data allocations.

The most common use for the remote storage class is to declare very large arrays:

```
remote double bigarray [10000];
```

This declaration reserves 80,000 bytes of storage for the 10,000 element double array. The linker organizes data memory for the process using bigarray as follows:



The remote storage class should be used sparingly, since the code required to access such variables is larger and slower than accessing non-remote variables.

As an alternative to using remotes, a program could declare a pointer to a large array and perform a memory request to the system to obtain the necessary storage.

The syntax for the remote storage class is identical to that of the extern storage class:

```
remote <data type> <variable>
```

The scope of the declared variable can be limited to the current file by specifying:

```
static remote <data type> <variable>
```

As an alternative to the remote storage class, a compile option is available to cause the compiler to always generate 32- or 16-bit offsets. c68 uses 16-bit offsets (by default) to access the data area: offset(A6). c68020 by default uses 32-bit offsets: (offset.l,A6). This can be controlled explicitly with the –k option:

Option:	Description:
cc –k0l	uses c68 with 32-bit data offsets
cc –k2w	uses c68020 with 16-bit data offsets

You should use care when compiling program sections separately as linking conflicts can arise if data references and/or definitions do not match among all sections.

PC-relative references generated by c68 are always 16-bit. If a reference does not reach the destination, the linker substitutes a reference to a process local **jump table** that contains the absolute address of the destination.

PC-relative references generated by c68020 are always 32-bit. No jump table reference is required as any destination address is addressable.

The Direct Storage Class

The C Compiler also supports the direct storage class. This class is not implemented by the 68000 C Compiler, but is available to provide portability from the 6809 to the 68000.

Control Character Escape Sequences

The escape sequences for non-printing characters in character constants and strings (see K&R on **character and string constants**) are extended as follows:

```
linefeed (LF):  \l
```

This is to distinguish LF (hex 0A) from \n which on OS-9 is the same as \r (hex 0D).

bit patterns:	\NNN	(octal constant)
	\dNNN	(decimal constant)
	\xNN	(hexadecimal constant)

For example, the following all have a value of 255 (decimal):

\377   \xff   \d255

System Calls and the Standard Library

The system interface supports almost all of the system calls of both OS-9 and UNIX. In order to facilitate the portability of programs from UNIX, some of the calls use UNIX names rather than OS-9 names for the same function. Even though the names are the same, consult the discussion of the function for any possible differences between the UNIX system call and the OS-9 equivalent.

There are a few UNIX calls that do not have exactly equivalent OS-9 calls. In these cases, the library function simulates the function of the corresponding UNIX call. In cases where there are OS-9 calls that do not have UNIX equivalents, the OS-9 names are used. Details of the calls and a name cross-reference are provided in the C Library Section of this manual.

The Standard Library

The C compiler includes a very complete library of standard I/O functions provided with most UNIX or UNIX-like systems. It is essential for any program which uses the high-level I/O functions from the standard library (such as fopen, getc, putc, etc.) to include the statement:

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

Chapter 3

Compiler Organization

Compiler Organization

The C compiler is made up of three main parts:

cpp	the preprocessor
c68, c68020	the compiler
o68	the object code improver

The macro assembler (r68, r68020) and linker (l68) are required to assemble and link the compiler output.

The C executive (cc or cc68) provides a uniform command interface between the user’s command line and each individual compiler phase.

The 68020 version of the compiler (c68020) executes on either a 68000 or 68020 system. It always generates 68020 cod. ec68 always generates 68000 code.

It is the the preprocessor’s job to condition the source program for the compiler. Conditioning involves gathering and expanding macros, including the contents of other files into the output, removing comments and providing the compiler with information required to report the files and lines in error (if any).

The compiler then takes the preprocessor output and (in a single pass) translates the source lines into assembly source code suitable for the macro assembler.

Any errors in the source program are displayed on the standard error path with the associated source line and a circumflex (^) pointing to the item causing the error. For example:

```
“prog.c”, line 10: **** undeclared variable ****
    i = j + 1;
    ^
```

Due to the one-pass nature of the compiler, you can invoke the object code improver to **clean-up** any inefficient code left in the assembly source by the compiler. This function identifies and removes needless instruction sequences. It recognizes and compresses dynamically identical program logic, changes long displacement relative branches to short where possible, and removes comment lines.

Object Code Output

The C compiler produces assembly code that is normally position independent, re-entrant and ROMable, assuming good programming techniques are employed. It is possible, by the very nature of the C language, to write code that modifies itself or accesses memory by absolute address. You should avoid this practice to be consistent with the OS-9 position independent shared module philosophy.

After the assembly code is assembled and linked, the output of the linker is in the form of a standard OS-9 executable machine language memory module (including module header and CRC check value).

The Cstart Routine

The C compiler system depends on a short code section called cstart.r as the first section of every program. It includes:

- the assembler directives to create an executable program module
- code to convert the parameter string passed by os9exec() into the argv and envp strings for main()
- code to initialize the C I/O facility
- other initialization code
- a call to the function main()

Although the compiler is intended to produce code for OS-9-based target systems, it is possible to alter the cstart routine for non-OS-9 or stand alone systems. The assembler source code for this routine is provided in a file called cstart.a. If you use the compiler to produce programs for non-OS-9 target systems you will also need to provide your own versions of the standard library functions that perform I/O, memory management, etc.

Run-Time Arithmetic Error Handling

K&R leave the treatment of various arithmetic errors open, merely saying that it is machine dependent. This implementation deals with a limited number of error conditions in a special way.

Integer division by zero causes a trap through the **divide by zero** machine vector. This causes a program to exit with the error status code of E$ZerDiv (000:105) unless a handler for this exception is provided.

All floating point errors, including division by zero, cause a trap through the trapv instruction machine vector. This causes the program to exit with the error status code of E$TrapV (000:107). These and other machine exceptions may be caught with a user-supplied assembly language routine.

Results of other possible errors are undefined.

The Stack

The upper part (higher addresses) of a C program's data area is reserved for the stack. Each procedure invocation uses stack space for linkage and automatic variable space.

It is impossible for the compiler to determine how much stack space a specific program requires. Programs that nest function calls very deeply or are highly recursive require more stack space. The default stack size is approximately 2K. You may increase the stack to any desired size using the –m compiler option flag.

The compiler generates code to check for stack underflow during procedure calls. Use the compiler's –s option flag to inhibit generation of stack checking code. Though the extra code produced makes slightly larger and slower programs, it is recommended that you retain the stack checking code until a program is well tested. Stack overflow bugs are among the most difficult to locate.

Interfacing to Assembly Language

C programs can run hand-written assembly language either by inline coding in C programs using the #asm and #endasm directives or by giving the name of previously assembled relocatable file(s) on the C executive command line.

It is very difficult to determine just where the compiler is during code generation, therefore it is recommended that assembly language code not be embedded within C functions. Assembly code is best placed outside of a function declaration or in a separately assembled module to be linked in with the rest of the program.

If c68 is being used, all functions and machine language subroutines are called by bsr instructions except functions more than 32k away.

The register usage conventions are (in general) as follows:

Register:	Usage:
D0 - D1	Function argument/return registers
D2 - D3	Compiler allocated temporaries
D4 - D7	Used for user register variables
A0 - A1	Compiler allocated temporaries
A2 - A4	Used for user register variables
A5	Frame Pointer
A6	Base address of variable storage area
A7	Stack Pointer

The compiler uses a complex register allocation method to provide the smallest, fastest code for the majority of programs encountered. The 68000 has a large number of processor registers. About half of these are made available to use as register variables. The rest are used by the compiler for storing intermediate results during the evaluation of expressions.

The A6 register (used as a pointer to the base of the global and static variables) is passed to a program when the program is forked and is never changed by C code.

The type of the argument and the order specified in the argument list indicates to the called function where the argument is: it is either in a register or on the stack.

For this discussion, an integral argument is an argument of type int, a pointer, or a char or short converted to an int. A double argument is an argument of type double or a float converted to a double.

A float is converted to a double before being passed on the stack.

The first integral argument is passed in d0, the second integral argument (if any) in d1. A single double argument is passed in d0 and d1, the most significant half in d0, the least significant half in d1. Any remaining arguments are pushed onto the stack. If the first argument is integral and the second is a double, the integral argument is passed in d0 and the double is passed entirely on the stack. Consult the examples of this method below.

If a function is to return a value, the integral (or float) value is returned in the d0.l register. A double value is returned in d0.l and d1.l.

Examples of C Argument Passing Techniques

Assumption:	C code:	Generated assembler:
init i, j, k; double a, b, c;	func(i);	move.l i,d0 bsr func
	func(i,j);	move.l j,d1 move.l i,d0 bsr func
	func(i,j,k);	move.l k,–(sp) move.l j,d1 move.l i,d0 bsr func
	func(a)	movem.l a,d0/d1 bsr func
	func(a,b)	move.l b+4,–(sp) move.l b+0,–(sp) movem.l a,d0/d1 bsr func
	func(a,i)	move.l i,–(sp) movem.l a,d0/d1 bsr func
	func(i,a)	move.l a+4,–(sp) move.l a+0,–(sp) move.l i,d0 bsr func

All functions (C or assembler) are required to restore any changed registers to the values they contained when the function was called. The only exceptions to this are function return register(s) and register(s) in which the function’s argument(s) are passed.

All parameters passed on the stack are 4-byte long words. Types char and short are sign extended to long words, types unsigned char and unsigned short are padded to the left with zeroes.

The 68000 bsr instruction, which is used for function calling, is limited to a ±32K address displacement. The bsr instruction has sufficient range for all but the very largest programs. In order to permit function calls outside this range while retaining position independence of the code, the linker (automatically) builds a jump table of long addresses of function entry points outside the range of bsr.

This table resides in the data area and can be accessed by the symbol _jmptbl defined by the linker. When coding assembly language routines, all external functions should be accessed by the word length displacement form of bsr so the linker can change the bsr to a jump if the displacement is too distant.

Consult the **OS-9/68000 Assembler/Linker/Debugger Manual** for more information on the jump table.

Chapter 4

Using The Termcap Library

Using the Termcap Library

There are six termcap library functions:

tgetent()	tgetflag()	tgetnum()
tgetstr()	tgoto()	tputs()

These functions are kept in the library file termlib.l. For standardization purposes, termlib.l is located in the system LIB directory. They are explained in detail in the following pages.

The termcap library functions allow a program to access the termcap database and extract information about the capabilities of a terminal. The functions only extract the information. The calling program determines the capabilities to be used.

Important: Refer to Using Professional OS-9 for complete information on setting up your termcap database file.

`tgetent()` must be called before any of the others. `tgetent()` extracts the entry for the terminal named name and places the data into the buffer pointed to by bufptr. The size of this buffer must be at least 1024 characters and must remain intact for all subsequent calls to tgetnum(), tgetflag() and tgetstr().

The termcap file is located in the SYS directory on the first successful open of /dd/SYS/termcap, /h0/SYS/termcap or /d0/SYS/termcap.

When linking programs containing these functions, use –l=/dd/lib/termlib.l (or whatever the appropriate path is on your system).

The calling program must define the external variables PC_, BC, UP and ospeed. PC_ is used rather than PC due to conflicts with assembler register names. The termcap library functions merely reference the variables, not define the storage for them.

`tgetent()` requires approximately 1K of stack normally. It requires 2K of stack if the tc capability is used. Be sure to give the program more stack memory during linkage. tgetent() recognizes both \n (LF) and \r (CR) to allow sharing of the same termcap file on OS-9 and UNIX systems.

The value placed in ospeed is the OS-9 baud rate code which is not the same as the equivalent UNIX baud rate code. No padding is used for ospeed values outside of the 0 to 16 range.

Typically, a program looks in the environment for TERM to get the name of the terminal to use. This is an automatic way to let programs know the name of the current terminal. For more information concerning the shell environment, consult Using Professional OS-9.

Programs that use termcap usually use only a few of the defined capabilities. Not all programs use all the capabilities and some capabilities are not used by any programs. Determine which capabilities are required for the application and extract the capabilities into variables for quick access as shown in the following example:

```
#include <stdio.h>
#include <sgstat.h>
#include <termcap.h>

#define TCAPSLEN400

extern char *getenv();

char tcapbuf[TCAPSLEN];    /* buffer for extracted termcap strings */
                              /* must remain intact for entire program */

char PC_;     /* pad character */
char *BC;     /* backspace character string */
char *UP;     /* up cursor  */
short ospeed;              /* terminal speed */
char *CL,     /* clear screen */
     *CM,     /* cursor motion */
     *CE,     /* clear end-of-line */
     *CD,     /* clear end-of-display */
     *SO,     /* standout begin */
     *SE,     /* standout end */
     *HO;     /* cursor home */

short
    lines,    /* lines on screen */
    colms;    /* columns on screen */

/* function to write one character */
int tputc(c)
char c;
{
    return write(1, &c, 1);
}
```

```
/* function to write a display string */
touts(s)
char *s;
{
#ifdef brain_damaged
    while (*s) tputc(*s++);
#else
    write(1, s, strlen(s));
#endif
}

/* function to write a terminal control string */
putpad(str)
char *str;
{
    tputs(str, 1, tputc);
}

/*
    Program to demonstrate calls to termcap library functions. This
    program will display on any terminal with a proper termcap database entry.
*/
main()
{
    register char *term_type, *temp;
    auto char tcbuf[1024];/* buffer for tgetent */
/* must remain intact for all tgetstr, */
/* tgetflag and tgetnum calls */
    auto char *ptr;
    if ((term_type = getenv("TERM")) == NULL) {
        fprintf(stderr, "Environment variable TERM not defined!\n");
        exit(1);
    }
    if (tgetent(tcbuf, term_type) <= 0) {
        fprintf(stderr,"Unknown terminal type '%s'!", term_type);
        exit(1);    }


    ptr = tcapbuf;

/* get stuff we are interested in */
    if (temp = tgetstr("PC", &ptr)) PC_ = *temp;
    CL = tgetstr("cl", &ptr);
    CM = tgetstr("cm", &ptr);
    CE = tgetstr("ce", &ptr);
    CD = tgetstr("cd", &ptr);
    UP = tgetstr("up", &ptr);
    SE = tgetstr("se", &ptr);
    SO = tgetstr("so", &ptr);
    HO = tgetstr("ho", &ptr);
    lines = tgetnum("li");
    colms = tgetnum("co");
```

```
    if (lines < 1 || colms < 1) {
        fprintf(stderr, “Like wow, man. No lines or columns!\n”);
        exit(1);
    }

    if (!(HO && CE && CL && CM && UP)) {/* must be given or else... */
        fprintf(stderr,“Incomplete termcap entry\n”);
        exit(1);
    }

    if (ptr >= &tcapbuf[TCAPSLEN]) {
        puts(“Terminal description too big!\n”);
        exit(1);
    }

    putpad(HO);/* home cursor */
    putpad(CL);/* clear screen */
    putpad(tgoto(CM, 50, 5));/* spot cursor */
    touts(“Fuzzy Wuzzy Wuzza Bare”);
    putpad(tgoto(CM, 30, 10));
    touts(“Fuzzy Wuzzy Hadno Hare”);
    putpad(tgoto(CM, 20, 15));
    touts(“Fuzzy Wuzzy Wuzn’t Fuzzy Wuzzy”);
    putpad(tgoto(CM, 0, lines-1));/* ready to exit */
    exit(0);
}
```

Section 2

The C Standard Library

This section of the manual contains chapter 5 which describes the standard library provided with the Microware C Compiler. Chapters 6 through 8 contain functions indexes organized by the following:

- name
- category
- use

Chapter 5

The C Standard Library

The C Standard Library

The standard library provided with the Microware C Compiler consists of a collection of high-level I/O, convenience, and system-level functions.

The high-level I/O functions provide facilities that are normally considered part of other languages (for example, the format statement of Fortran). C technically does not have any I/O statements, relying instead on the standard library. This enhances the versatility and portability of the language.

In addition, automatic buffering of I/O paths improves the speed of file access because fewer calls to the host operating system are required.

The high-level I/O functions should not be confused with the low-level system calls with the same names (for example, fopen() and open()). The standard library functions maintain a structure (declared as FILE in <stdio.h>) for each open file. This structure holds status information for the file. A pointer (usually supplied by fopen()) to this structure is the **identity** of the file, and it is passed to the various I/O functions. The I/O functions make low-level system calls when appropriate.

Using a file pointer in a system call or a path number in a high-level I/O call is a common mistake among beginners to C and, if made, will at best crash your program, or at worst, provide unpredictable program behavior.

In addition to the C I/O functions, the standard library contains functions to perform character classification and conversion, string manipulation, memory management, mathematical functions, and system related operations.

Each function description includes a synopsis and details on using the function. The synopsis shows how the function and arguments would look if written as a C function definition, even if the actual function is a macro or is written in assembly language.

For example, the synopsis for fopen appears as follows:

```
#include <stdio.h>

FILE *fopen(name,action)
char *name, *action;
```

The synopsis indicates that the function fopen requires the header file <stdio.h>, returns a pointer to a structure of type FILE, and requires two arguments, both pointers to a character string. The argument names are suggestions only; you can use any name.

When an error occurs, C functions typically return an error code in the global variable errno. You must include the file <errno.h> in C programs so that errno will be declared.

ANSI Library Functions

asctime	clock	ctime	difftim
gmtime	localtime	memchar	memcmp
memcpy	memmove	memset	mktime
time			

Character Classification Functions

The character classification functions are really macros defined in <ctype.h>. Be careful when using these macros to avoid macro expansion side-effects. The macros provide a machine independent method of character classification:

isalnum	isalpha	isascii	iscntrl
isdigit	islower	isprint	ispunct
isspace	isupper	isxdigit	

Character Conversion Functions

The character conversion functions provide the ability to convert a string of characters to their numeric representation and to change the case of characters:

_atou	_tolower	_toupper	atof
atoi	atol	toascii	tolower
toupper			

Mathematical Functions

Transcendental and algebraic math functions provided in the Microware standard library are actually hooks into the OS-9 math trap handlers. If, for example, the math handler were replaced with a handler that accessed floating point hardware, the application program would require absolutely no changes to use the new trap handler. The math functions included in the standard library are those found on most UNIX systems:

abs	acos	asin	atan
ceil	cos	exp	fabs
floor	frexp	hypot	ldexp
log	log10	modf	pow
sin	sqrt	tan	

Memory Management Functions

The standard library provides for a suite of functions for requesting and freeing memory in a machine and operating system independent manner. Various flavors of UNIX contain different functions, but generally provide the same results. The memory management functions provided are:

_lcalloc	_lfree	_lmalloc	_lrealloc
_mallocmin	calloc	free	malloc
realloc	sbrk		

For special use with OS-9, additional functions are available for increased efficiency:

_freemem	_srqmem	_srtmem	ebrk
ibrk			

Miscellaneous Functions

Several miscellaneous functions appear in the standard library that relieve you from routine tasks:

_errmsg	_prgname	_strass	closedir
freemem	longjmp	mktemp	opendir
qsort	readdir	rewinddir	seekdir
setjmp	stacksiz	system	telldir

OS-9 System Functions

The standard library provides a large number of functions to directly access OS-9 system calls. The following functions are available to provide access to selected OS-9 system calls:

_cmpnam	_cpymem	_ev_creat
_ev_delete	_ev_info	_ev_link
_ev_pulse	_ev_read	_ev_set
_ev_setr	_ev_signal	_ev_unlink
_ev_wait	_ev_waitr	_get_module_dir
_get_process_desc	_get_process_table	_getsys
_gs_devn	_gs_eof	_gs_gfd
_gs_opt	_gs_pos	_gs_rdy
_gs_size	_julian	_mkdata_module
_parsepath	_prsnam	_setcrc
_setsys	_ss_attr	_ss_dcoff
_ss_dcon	_ss_dsrts	_ss_enrts
_ss_lock	_ss_opt	_ss_pfd
_ss_rel	_ss_rest	_ss_size
_ss_ssig	_ss_tiks	_ss_wtrk
_sysdate	_sysdbg	alm_atdate
alm_atjul	alm_cycle	alm_delete
alm_set	attach	chain
chainc	chxdir	crc
create	detach	getstat
intercept	makdir	make_module
modcload	modlink	modload
modloadp	munlink	munload
os9exec	os9fork	os9forkc
readln	setstat	sigmask
srqcmem	tsleep	unlinkx
writeln		

Standard I/O Functions

The following functions make up the standard I/O functions of the C library. The functions accept arguments, return values, and generally behave according to the K&R (and UNIX) library functions. All of these functions require that you include the <stdio.h> header file:

cleareof	clearerr	fclose	fdopen
feof	ferror	fflush	fgetc
fgets	fileno	fopen	fprintf
fputs	fread	freopen	fscanf
fseek	ftell	fwrite	getc
getchar	gets	getw	pffinit
pflinit	printf	putc	putchar
puts	putw	rewind	scanf
setbuf	sprintf	sscanf	ungetc

String Handling Functions

The C language does not have a character string data type. Instead, it stores strings as character arrays and the standard library provides functions to manipulate them:

findnstr	findstr	index	rindex
strcat	strcmp	strcpy	strhcpy
strlen	strncat	strncmp	strncpy

Terminal Manipulation Functions (Termcap)

tgetent	tgetflag	tgetnum	tgetstr
tgoto	tputs		

UNIX-like System Functions

Because many C programs are written for the UNIX operating system, numerous UNIX functions are used in those programs. To help provide portability, many of those functions are supported by this compiler. Most of them have very similar equivalent system calls in OS-9, others are emulated. Some UNIX system calls cannot be conveniently emulated and do not appear in the library.

Even though the system call has the same name as the UNIX equivalent and performs essentially the same operation, check the discussion of the function for any subtle differences (like the mode values on open()).

The following are the UNIX-like system calls appearing in the standard library. The function descriptions for these functions categorize them as being UNIX System functions:

_exit	access	chdir	chmod
chown	close	creat	dup
exit	getenv	getime	getpid
getuid	kill	lseek	mknod
open	pause	prerr	read
setime	setpr	setuid	sleep
unlink	wait	write	

_atou()

Alpha to Unsigned Conversion

Synopsis

```
unsigned _atou(string)
char    *string;
```

Function

_atou() converts a string into its appropriate unsigned numeric value, if possible. string points to a string that contains a printable representation of a number expressed in the following format: [+/–]<digits>. _atou() treats long and int values identically.

_cmpnam()

Compare Two Strings

Synopsis

```
int _cmpnam(target, pattern, patlen)
char *target,              /* pointer to target string */
     *pattern;             /* pointer to string pattern for comparison */
int  patlen;               /* length of pattern string */
```

Function

_cmpnam() performs a name comparison using the OS-9 system call F$CmpNam. _cmpnam() compares the target string to the pattern string to determine if they are the same. The target name must be terminated by a null byte. Upper and lower case are considered to match.

Two metacharacters are recognized in the pattern string: a question mark (?) matches any single character and an asterisk (*) matches any string of characters.

_cmpnam() returns 0 if the strings match. –1 is returned if no match occurs.

See Also

F$CmpNam in the OS-9 Technical Manual.

_cpymem()

Copy External Memory

Synopsis

```
int _cpymem(pid, count, from, into)
short pid;    /* process ID */
int  count;   /* number of bytes to copy */
char *from,   /* pointer to memory to copy */
     *into;   /* pointer to copy buffer */
```

Function

`_cpymem()` copies memory owned by another process (or the system) into the buffer pointed to by into. from is the address in the process's address space from which to copy. `pid` is the process ID number of the external process. If pid is zero, the system's address space is assumed. count is the number of bytes to copy. `_cpymem` returns –1 if an error occurs. The appropriate error code is placed in the global variable errno.

See Also

`F$CpyMem` in the OS-9 Technical Manual.

_errmsg()

Print an Error Message

Synopsis

```
int _errmsg(nerr, msg[, arg1, arg2, arg3])
int  nerr;    /* error number */
char *msg;    /* pointer to error message */
```

Function

`_errmsg()` displays an error message on the standard error path along with the name of the program. The message string msg is displayed in the following format:

```
prog: <message text>
```

Important: `prog` is the module name of the program and <message text> is the string passed in msg.

For added flexibility in message printing, the msg string can be a conversion string suitable for fprintf() with up to three additional arguments of any integral type. `nerr` is returned as the value of the function so _errmsg() can be used as a parameter to a function such as exit() or prerr().

Example

Assume the program calling the function is named foobar:

Call: `_errmsg(1,"programmed message\n");`

Prints: `foobar: programmed message`

Call: `exit(_errmsg(errno,"unknown option '%c'\n",'q'));`

Prints: `foobar: unknown option 'q'`

Then exits with error code in errno.

See Also

`fprintf()`, `_prgname()`

_ev_creat()

Create Event

Synopsis

```
#include <events.h>

int _ev_creat(ev_value, wait_inc, signal_inc, ev_name)
int  ev_value,             /* initial value of event */
     wait_inc,             /* event wait increment */
     signal_inc;           /* event occurrence increment */
char *ev_name;             /* pointer to event name */
```

Function

`_ev_creat()` creates an event. ev_name is a pointer to a string containing the name of the event. `ev_value` is the initial value for the event. wait_inc and signal_inc are increments applied to the event each time the event occurs or is waited for. An event ID number is returned if the event is successfully created.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_delete()

Delete Event

Synopsis

```
#include <events.h>

int _ev_delete(ev_name)
char *ev_name;              /* pointer to event name */
```

Function

`_ev_delete()` deletes an event. ev_name is a pointer to a string containing the name of the event. The use count for the event must be zero before the event can be deleted.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_info()

Obtain Event Information

Synopsis

```
#include <events.h>

int _ev_info(ev_index, ev_buffer)
int   _ev_index;           /* place to begin search in event table */
event *ev_buffer;          /* pointer to buffer for event information */
```

Function

`_ev_info()` returns information about an event. The event table is indexed from zero to one less than the maximum number of events allowed on the system. `ev_index` corresponds to the starting point in the event table to search for an event. `ev_buffer` is a pointer to the event struct buffer used to hold the event information if found.

`-1` is returned if ev_index is greater than all active events in the table and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_link()

Link to Existing Event

Synopsis

```
#include <events.h>

int _ev_link(ev_name)
char *ev_name;             /* pointer to event name */
```

Function

`_ev_link` links to an existing event. `ev_name` is a pointer to a string containing the name of the event. An event ID number is returned if the event is successfully linked.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_pulse()

Signal Event Occurrence

Synopsis

```
#include <events.h>

int _ev_pulse(ev_id, ev_value, allflag)
int   ev_id,              /* event ID */
      ev_value;           /* value to which event is set */
short allflag;            /* if 0x0000: first waiting proc is activated */
                          /*    0x8000: all waiting procs are activated */
```

Function

`_ev_pulse()` indicates that an event has occurred. `ev_id` is the event ID returned from _ev_creat() or _ev_link(). The event variable is set to the value given by ev_value, the normal signal increment is not applied. The normal event value is restored after activating processes, if any. If allflag is zero, the first process waiting for the event is activated. If allflag is 0x8000, all processes waiting for the event that have a value in range are activated.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_read()

Read Event Without Waiting

Synopsis

```
#include <events.h>

int _ev_read(ev_id)
int ev_id;    /* event ID */
```

Function

`_ev_read()` reads the value of an event without waiting or affecting the event variable. `ev_id` is the desired event ID number.

If an error occurs, –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_set()

Set Event Variable and Signal Event Occurrence

Synopsis

```
#include <events.h>

int _ev_set(ev_id, ev_value, allflag)
int   ev_id,              /* event ID */
      ev_value;           /* value to which event is set */
short allflag;            /* if 0x0000: first waiting proc is activated */
                          /*    0x8000: all waiting procs are activated */
```

Function

`_ev_set` indicates that an event has occurred. `ev_id` is the event ID returned from _ev_creat() or _ev_link(). The event variable is set to the value given by ev_value, and the normal signal increment is not applied. Processes waiting for the event are then activated. If allflag is zero, the first process waiting for the event is activated. If allflag is 0x8000, all processes waiting for the event that have a value in range are activated.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_setr()

Set Relative Event Variable and Signal Event Occurrence

Synopsis

```
#include <events.h>

int _ev_setr(ev_id, ev_value, allflag)
int   ev_id,              /* event ID */
      ev_value;           /* value to increment event */
short allflag;            /* if 0x0000: first waiting proc is activated */
                          /*    0x8000: all waiting procs are activated */
```

Function

`_ev_setr()` indicates that an event has occurred. ev_id is the event ID returned from _ev_creat() or _ev_link(). The event variable is incremented by the value given by ev_value. Processes waiting for the event are then activated. If allflag is zero, the first process waiting for the event is activated. If allflag is 0x8000, all processes waiting for the event that have a value in range are activated.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_signal()

Signal Event Occurrence

Synopsis

```
#include <events.h>

int _ev_signal(ev_id, allflag)
int ev_id;                  /* event ID */
short allflag;              /* if 0x0000: first waiting proc is activated */
                            /*    0x8000: all waiting procs are activated */
```

Function

`_ev_signal()` indicates that an event has occurred. ev_id is the event ID returned from _ev_creat() or _ev_link(). The current event variable is updated by the signal increment (given when the event was created). If allflag is zero, the first process waiting for the event is activated. If allflag is 0x8000, all processes waiting for the event that have a value in range are activated.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_unlink()

Unlink Event

Synopsis

```
#include <events.h>

int _ev_unlink(ev_id)
int ev_id;                    /* event ID */
```

Function

`_ev_unlink()` informs the system that the event is no longer required by this process. The link count of the event is decremented. If the event count becomes zero, the event is not deleted. To delete an event with a link count of zero, use _ev_delete() .

`ev_id` is the event ID number.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_wait()

Wait for Event

Synopsis

```
#include <events.h>

int _ev_wait(ev_id, ev_min, ev_max)
int ev_id,                /* event ID */
    ev_min,               /* minimum range value for event */
    ev_max;               /* maximum range value for event */
```

Function

`_ev_wait()` waits for an event to occur. `ev_id` gives the ID of the event. The event value is compared to the range values given by ev_min and ev_max. If the event value is not in the specified range, the process waits until some other process places the value within the range. Once in range, the wait increment is applied to the event value. The actual event value is returned as the value of the function.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_ev_waitr()

Wait for Relative Event

Synopsis

```
#include <events.h>

int _ev_waitr(ev_id, ev_min, ev_max)
int ev_id,                /* event ID */
    ev_min,               /* minimum range value for event */
    ev_max;               /* maximum range value for event */
```

Function

`_ev_waitr` waits for an event to occur. ev_id gives the ID of the event. The event value is compared to the range values given by ev_min and ev_max. The current event value is added to the range values before the comparison. If the event value is not in the specified range, the process waits until some other process places the value within the range. Once in range, the wait increment is applied to the event value. The actual event value is returned as the value of the function.

`-1` is returned if an error occurs and the appropriate error code is placed in the global variable errno.

See Also

`F$Event` in the OS-9 Technical Manual.

_exit()

Task Termination

Synopsis

```
_exit(status)

int status;   /* exit status. if 0:  normal exit */
              /*                else = error code  */
```

Function

`_exit()` causes immediate termination of a program. status provides an indication to the parent process as to the success or failure of the program. An exit status of zero is considered normal termination; a non-zero value is interpreted as an error code by most programs (especially the shell). This function never returns.

Another form of this call, exit(), flushes I/O buffers and basically cleans up after the program before exiting.

See Also

`F$Exit` in the OS-9 Technical Manual.

_freemin()

Set Memory Reclamation Bound

Synopsis

```
_freemin(size)

int size;     /* minimum # of bytes to return */
```

Function

`_freemin()` allows you to set the minimum size for a free block of memory or concatenation of free blocks to be returned to the operating system by free(). If a program is known to **hog** memory, you can instruct free() never to return memory to the operating system.

If size is positive, it is used as the minimum number of bytes for a free block of memory to be a candidate for return to the operating system.

If size is negative, free() never tries to return memory allocated by malloc() to the system.

It is unnecessary to invoke _freemin() to return memory to the system. The default minimum size to be returned is 4K. Any size smaller than 4K specified by _freemin() is ignored.

`_freemin()` returns no value of interest.

See Also

`free()`, `ebrk()`, `ibrk()`, `sbrk()`, `_freemin()`

_get_module_dir()

Get Module Directory Entry

Synopsis

```
#include <module.h>

int _get_module_dir(buffer, count)
char *buffer;              /* pointer to buffer for module dir info */
int  count;                /* number of bytes to copy into buffer */
```

Function

`_get_module_dir()` copies the system's module directory into the buffer pointed to by buffer for inspection. A maximum of count bytes are copied. The number of bytes actually copied is returned.

If an error occurs, –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

`F$GModDr` in the OS-9 Technical Manual.

_get_process_desc()

Get Process Descriptor Copy

Synopsis

```
#include <procid.h>

int _get_process_desc(pid, count, buffer)
short  pid;                /* process ID */
int    count;              /* number of bytes to copy into buffer */
procid *buffer;            /* pointer to buffer for descriptor info */
```

Function

`_get_process_desc()` copies a process descriptor into the buffer pointed to by buffer for inspection. `pid` is the process ID number of the desired process. A maximum of count bytes are copied.

If an error occurs, –1 is returned and the appropriate error code is placed in the global variable errno.

Example Call:

```
procid pbuf;
_get_process_desc(pid,sizeof(pbuf),&pbuf);
```

See Also

`F$GPrDsc` in the OS-9 Technical Manual.

_get_process_table()

Get Process Table Entry

Synopsis

```
#include <procid.h>

int _get_process_table(buffer, count)
char *buffer;              /* pointer to buffer for descriptor info */
int count;                 /* number of bytes to copy into buffer */
```

Function

`_get_process_table()` copies the system's process descriptor table into the buffer pointed to by buffer for inspection. A maximum of count bytes are copied. The number of bytes actually copied is returned.

If an error occurs, –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

The discussion of F$GPrDBT in the OS-9 Technical Manual.

_get_sys()

Get System Global Variables

Synopsis

```
#include <setsys.h>

int _getsys(glob, size)
short glob;                /* offset to global variable */
int  size;                 /* size of global variable */
```

Function

`_getsys()` examines a system global variable. These variables are defined in the sys.l system library file. The same values are defined in <setsys.h> for C programs. Each of these variables begin with a D_ prefix. glob is the offset to the desired variable. size is the size of the variable.

`_getsys()` returns the value of the variable if the examine request succeeds. If the request fails, –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

The discussion of _setsys(); the discussion of F$SetSys in the OS-9 Technical Manual.

_gs_devn()

Get Device Name

Synopsis

```
int _gs_devn(path, buffer)
int path;                   /* path number */
char *buffer;               /* pointer to buffer for device name */
```

Function

You can determine the name of the device open on a path by _gs_devn(). path is a path number of an open path and buffer is a pointer to a character array into which the null-terminated device name is placed.

If the path number is invalid, the function returns –1 as its value and the appropriate error code is placed in the global variable errno.

Caveats

Be sure to reserve at least 32 characters to receive the device name.

See Also

`I$GetStt` in the OS-9 Technical Manual.

_gs_eof()

Check for End-of-File

Synopsis

```
int _gs_eof(path)
int path;                  /* path number */
```

Function

`_gs_eof()` determines if the file open on path is at end-of-file. If it is at end-of-file, the value 1 is returned. If it is not at end-of-file, 0 is returned.

If path is invalid, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

`_gs_eof()` cannot determine end-of-file on SCF devices. SCF devices return an E$EOF error when the EOF character is read. DO NOT use this call if you are using the buffered I/O facility on the path. Instead, use feof().

See Also

`feof()`; `I$GetStt` in the OS-9 Technical Manual.

_gs_gfd()

Get File Descriptor Sector

Synopsis

```
#include <direct.h>

int _gs_gfd(path, buffer, count)
int    path;              /* path number */
struct fildes *buffer;    /* pointer to buffer for descriptor info */
int    count;             /* max number of bytes to copy into buffer */
```

Function

`_gs_gfd` places a copy of the RBF file descriptor sector of the file open on path into the buffer pointed to by buffer. A maximum of count bytes are copied. The structure fildes declared in <direct.h> provides a convenient means to access the file descriptor information.

If an error occurs, the function returns the value –1 and the appropriate error value is placed in the global variable errno.

Caveats

Be sure the buffer is large enough to hold count bytes. This call is effective only on RBF devices. Declaring the buffer as type struct fildes is perfectly safe as this structure is predefined to be large enough for the file descriptor.

See Also

`_ss_pfd()`; `I$GetStt` in the OS-9 Technical Manual.

_gs_opt()

Get Path Options

Synopsis

```
#include <sgstat.h>

int _gs_opt(path, buffer)
int    path;              /* path number */
struct sgbuf *buffer;     /* pointer to buffer for path desc info */
```

Function

`_gs_opt()` copies the options section of the path descriptor open on path into the buffer pointed to by buffer. `sgbuf` provides a convenient means to access the individual option values. `sgbuf` is declared in <sgstat.h>

If the path is invalid, _gs_opt() returns the value –1 and the appropriate error code is placed in the variable errno.

Caveats

Be sure the buffer is large enough to hold the options. Declaring the buffer as type struct sgbuf is perfectly safe as this structure is predefined to be large enough for the options.

See Also

`_ss_opt()`; `I$GetStt` in the OS-9 Technical Manual.

_gs_pos()

Get Current File Position

Synopsis

```
int _gs_pos(path)
int path;                  /* path number */
```

Function

`_gs_pos()` returns the value of the file pointer for the file open on path.

If the path is invalid or the device is not an RBF device, –1 is returned as the function value and the appropriate error code is placed in the global variable errno.

Caveats

This call is effective only on RBF devices. It is unique to OS-9; the equivalent portable call is lseek(). DO NOT use this call if buffered I/O is being performed on the path; instead use ftell().

See Also

`lseek()`; `I$GetStt` in the OS-9 Technical Manual.

_gs_rdy()

Test for Data Available

Synopsis

```
int _gs_rdy(path)
int path;                    /* path number */
```

Function

`_gs_rdy()` checks the SCF device open on path to see if data is waiting to be read. Read requests to OS-9 wait until enough bytes have been read to satisfy the byte count given, thereby suspending the program until the read is satisfied.

A program can use this function to determine the number of bytes, if any, waiting to be read, and then issue a read request for only the number of bytes actually received.

If the path is invalid, no data is available to be read, or the device is not a SCF device, a value of –1 is returned as the function value and the appropriate error code is placed in the global variable errno. Otherwise, the number of bytes available to be read is returned.

Caveats

This call is effective only on SCF or pipe devices. This function is not intended for use with the buffered I/O calls (like getc); unpredictable results may occur. Use low-level functions when using _gs_rdy().

See Also

`read()`, `readln()`; `I$GetStt` in the OS-9 Technical Manual.

_gs_size()

Get Current File Size

Synopsis

```
int _gs_size(path)
int path;                    /* path number */
```

Function

`_gs_size()` determines the current size of the file open on path. If the path is invalid or the device is not a RBF device, a value of –1 is returned as the function value and the appropriate error code is placed in the global variable errno. Otherwise, the size of the file is returned.

Caveats

This call is effective only on RBF devices.

See Also

`I$GetStt` in the OS-9 Technical Manual.

_julian()

Convert Date/Time to Julian Value

Synopsis

```
int _julian(time, date)

int *time,    /* pointer to time value */
    *date;    /* pointer to date value */
```

Function

`_julian()` converts the time and date from standard OS-9 format to the Julian equivalents. Note that time and date are pointers to the OS-9 format values. The following format is assumed:

	byte 0	byte 1	byte 2	byte 3
time:	0	hour (0–23)	minute	second
date:	year (2–bytes)		month	day

`_julian()` modifies the objects to which its arguments point as follows:

time:	Seconds since midnight (0–86399)
date:	Julian Day Number

If an error occurs, a value of –1 is returned as the function value and the appropriate error code is placed in the global variable errno.

Example

```
main()

{
   int date,time,tick;
   short day;

   _sysdate(0,&time,&date,&day,&tick);
   _julian(&time,&date);

   printf("The Julian date is %d. \
        Cinderella dies in %d seconds!\n",date,86400-time);
}
```

Caveats

Be careful to pass pointers for date and time values.

See Also

`_sysdate()`; `F$Time`, `F$Julian` in the OS-9 Technical Manual.

_lcalloc()

Allocate Storage for Array (Low-Overhead)

Synopsis

```
void *_lcalloc(nel, elsize)
unsigned long nel,         /* number of elements in array */
              elsize;      /* size of elements */
```

Function

This function allocates space for an array. nel is the number of elements in the array, and elsize is the size of each element. The allocated memory is cleared to zeroes.

This function calls _lmalloc() to allocate memory. If the allocation is successful, _lcalloc() returns a pointer to the area. If the allocation fails, _lcalloc() returns zero (NULL).

Important: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the 4-byte check value.

Caveats

Extreme care should be used to insure that only the memory assigned is accessed. Modifying addresses immediately above or below the assigned memory causes unpredictable program results.

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory. (Note the _lfree() and _lrealloc() parameters.)

See Also

`_srqmem(), _srtmem(), _lfree(), _lmalloc(), _lrealloc()`

_lfree()

Return Memory (Low-Overhead)

Synopsis

```
void _lfree(ptr, size)
void            *ptr;      /* pointer to memory to be returned */
unsigned long   size;      /* size of memory to be returned */
```

Function

`_lfree()` returns a block of memory granted by _lcalloc() or _lmalloc(). The memory is returned to a pool of memory for later re-use by _lcalloc() or _lmalloc().

`_lfree()` never returns an error.

Important: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the 4-byte check value.

Caveats

If _lfree() is used with something other than a pointer returned by _lmalloc() or _lcalloc(), the memory lists maintained by _lmalloc() will be corrupted and programs may behave unpredictably.

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory.

See Also

`_lcalloc()`, `_lmalloc()`, `_lrealloc()`

_lmalloc()

Allocate Memory from an Arena (Low-Overhead)

Synopsis

```
void *_lmalloc(size)
unsigned long   size;      /* size of memory block to allocate */
```

Function

`_lmalloc()` returns a pointer to a block of memory of size bytes. The pointer is suitably aligned for storage of data of any type.

`_lmalloc()` maintains an amount of memory called an **arena** from which it grants memory requests. `_lmalloc()` will search its arena for a block of free memory large enough for the request and, in the process, coalesce adjacent blocks of free space returned by the _lfree() function. If sufficient memory is not available in the arena, _lmalloc() calls _srqmem() to get more memory from the system.

`_lmalloc()` returns zero (NULL) if there is no available memory or if the arena is detected to be corrupted.

Important: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the 4-byte check value.

Caveats

Extreme care should be used to insure that only the memory assigned by _lmalloc() is accessed. Modifying addresses immediately above or below the assigned memory or passing _lfree() a value not assigned by _lmalloc() causes unpredictable program results.

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory. (Note the _lfree() and _lrealloc() parameters.)

See Also

`_lcalloc()`, `_lfree()`, `_lrealloc()`

_lrealloc()

Resize a Block of Memory (Low-Overhead)

Synopsis

```
void *_lrealloc(oldptr, newsize, oldsize)
void          *oldptr;     /* old pointer to block of memory */
unsigned long newsize;     /* size of new memory block */
              oldsize;     /* size of old memory block */
```

Function

`_lrealloc()` re-sizes a block of memory pointed to by oldptr. oldptr should be a value returned by a previous _lmalloc(), _lcalloc() or _lrealloc().

`_lrealloc()` returns a pointer to a new block of memory. The size of this new block is specified by newsize. The pointer is aligned to store data of any type.

If newsize is smaller than oldsize, the contents of the old block are truncated and placed in the new block. Otherwise, the entirety of the old block's contents begin the new block.

The results of _lrealloc(NULL,newsize,0) and _lmalloc(size) are the same.

`_lrealloc()` returns zero (NULL) if the requested memory is not available or newsize is specified as zero.

Important: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the 4-byte check value.

Caveat

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory.

See Also

`_freemin()`, `_lcalloc()`, `_lfree()`, `_lmalloc()`

_mallocmin()

Set Minimum Allocation Size

Synopsis

```
_mallocmin(size)
unsigned size;             /* minimum allocation size in bytes */
```

Function

`_mallocmin()` sets the minimum amount of memory that allocation functions may request through srqmem(). The size parameter cannot be less than the system memory block size. If a smaller size is requested, size is automatically set to the system memory block size.

OS-9 allows each process only 32 different memory segments; therefore, size should be increased if a program requires a great amount of memory. The extra space may be necessary if memory is fragmented.

`_mallocmin()` never returns an error.

See Also

`_lcalloc()`, `_lmalloc()`, `_lrealloc()`, `calloc()`, `malloc()`, `realloc()`

_mkdata_module()

Create a Data Memory Module

Synopsis

```
#include <module.h>

char *_mkdata_module(name, size, attr, perm)
char     *name;            /* pointer to name of module */
unsigned size;             /* size of module in bytes */
short    attr,             /* module attribute/revision */
         perm;             /* module access permissions */
```

Function

`_mkdata_module()` creates a data memory module. Other processes on the system can then access the data module by modlink() . name is the desired name of the module. `size` is the size in bytes of the module. attr is the attribute/revision word, and perm is the access permission word. The memory in the data module is initially cleared to zeroes.

Important: The size value does not include the module header and CRC bytes. The size given is the amount of memory available for actual use.

`_mkdata_module()` returns a pointer to the beginning of the module header.

If the data module cannot be created, a value of –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

`F$DatMod` in the OS-9 Technical Manual.

_parsepath()

Parse Disk File (RBF) Pathlist

Synopsis

```
int _parsepath(string)
char *string;              /* pointer to pathlist */
```

Function

`_parsepath()` parses a disk file (RBF) pathlist. string is a pointer to a pathlist. This function is useful for programs that must determine the validity of a pathlist name without actually creating or opening a file.

`_parsepath()` returns the number of bytes in the valid pathlist. If RBF does not accept the pathlist as valid, _parsepath() returns –1.

See Also

`_prsnam()`; `F$PrsNam` in the OS-9 Technical Manual.

_prgname()

Get Module Name

Synopsis

```
char *_prgname()
```

Function

`_prgname()` returns a pointer to the name of the module being executed. Normally, the argv[0] string indicates the name of the program as invoked by the parent process via os9exec(). You can use _prgname() to determine the actual name of the module as it appears in the module directory.

Caveats

If the code that calls this function is executing in a trap handler, the name of the trap handler module is returned.

See Also

`_errmsg()`

_prsnam()

Parse Path Name Segment

Synopsis

```
int _prsnam(string)
char *string;             /* pointer to path name segment */
```

Function

_prsnam() parses a path name segment using the F$PrsNam system call. string is a pointer to a path segment.

_prsnam() returns the number of characters in a valid path name segment. If the path name segment is invalid, _prsnam() returns –1. You can use _parsenam() to determine if a path is a valid disk file (RBF) pathlist.

You can use successive calls to _prsnam() to parse a complete path name.

See Also

_parsepath(); F$PrsNam in the OS-9 Technical Manual.

_setcrc()

Re-Validate Module CRC

Synopsis

```
#include <module.h>

int _setcrc(module)
mod_exec *module          /* pointer to executable memory module */
```

Function

_setcrc() updates the header parity and CRC of a module in memory. The module must have correct size and sync bytes; other parts of the module are not checked.

_setcrc() returns –1 and the appropriate error code in the global variable errno if an error occurs.

See Also

F$SetCRC in the OS-9 Technical Manual.

_setsys()

Set/Examine System Global Variables

Synopsis

```
#include <setsys.h>

int _setsys(glob, size[, value])
short glob;                /* global variable */
int   size;                /* size of global variable */
[int  value;]              /* if given, value to set variable */
```

Function

Use _setsys() to change or examine a system global variable. These variables are defined in LIB/sys.l. The same values are defined in <setsys.h> for use by C programs. These variables begin with a D_ prefix. glob is the offset to the desired variable. `size` is the size of the variable. `size` & 0x80000000 is used to examine the variable. `value` is an optional argument used only when changing a variable.

`_setsys()` returns the value of the variable on an examination request. On a change request, _setsys() returns the value of the variable before the change.

`_setsys()` returns –1 and the appropriate error code in the global variable errno if the examine or change request fails.

Important: Use _getsys() to examine variables without the possibility of accidental change.

See Also

`_getsys()`; `F$SetSys` in the OS-9 Technical Manual.

_srqmem()

System Memory Request

Synopsis

```
char *_srqmem(size)
unsigned size;            /* requested number of bytes */
```

Function

When tight control over memory allocation is required, _srqmem() and the complementary function _srtmem() are provided to request and return system memory.

This function is a direct hook into the OS-9 F$SRqMem system call. The specified size is rounded to a system-defined block size. A size of 0xffffffff obtains the largest contiguous block of free memory in the system. The global unsigned variable _srqsiz may be examined to determine the actual size of the block allocated.

If successful, a pointer to the memory granted is returned. If the request was not granted, _srqmem() returns the value –1 and the appropriate error code is left in the global variable errno.

The pointer returned always begins on an even byte boundary. Take care to preserve the value of the pointer if the memory is to be returned via _srtmem().

Caveats

The F$SrqMem request is actually intended for system level use, but the extended addressing range of the 68000 required some method to obtain memory without regard to where the memory is physically located.

A user process may have up to 32 non-contiguous F$SrqMem requests active at a given time. Ideally, the requests should be as large as practical, and preferably some multiple of 1K.

See Also

sbrk(), ibrk(), ebrk(), _srtmem(), malloc(), free()

_srtmem()

System Memory Return

Synopsis

```
int _srtmem(size, ptr)
unsigned size;              /* number of bytes to return */
char *ptr;                  /* pointer to memory to return */
```

Function

`_srtmem()` is a direct hook into the OS-9 F$SRtMem system call. It is used to return memory granted by _srqmem(). Care should be taken to ensure that size and ptr are the same as those returned by _srqmem().

If an error occurs, the function returns the value –1 and the appropriate error code is placed in the global variable errno.

See Also

`sbrk()`, `ibrk()`, `ebrk()`, `_srqmem()`, `malloc()`, `free()`

_ss_attr()

Set File Attributes

Synopsis

```
int _ss_attr(path, attr)
int   path;                /* path number */
short attr;                /* file attributes to set */
```

Function

`_ss_attr()` changes a disk file's attributes. `_gs_gfd()` determines the current attributes of a file. Only the owner of the file or the super user can change the attributes of a file.

The attributes selected in the word attr are set in the file open on path. The header file <modes.h> defines the valid mode values.

If an error occurs, _ss_attr() returns –1 as its value and the appropriate error code is placed in the global variable errno.

Caveats

This function is effective even if the owner or super user does not have write permission to the path. It is not permitted to set the directory bit of a non-directory file or to clear the directory bit of a directory that is not empty.

See Also

`_gs_gfd()`, `_ss_pfd()`; `I$SetStt` in the OS-9 Technical Manual.

_ss_dcoff()

Send Data Carrier Lost Signal to Process

Synopsis

```
int _ss_dcoff(path, sigcode)
int   path;                /* path number */
short sigcode;             /* signal code to send */
```

Function

`_ss_dcoff()` sends a signal (sigcode) to the calling process when the “data carrier detect line” associated with the device is lost. –1 is returned and errno set if unable to register the signal.

See Also

`I$SetStt` in the OS-9 Technical Manual.

_ss_dcon()

Send Data Carrier Present Signal to Process

Synopsis

```
SYNOPSIS: int _ss_dcon(path, sigcode)
int   path;                /* path number */
short sigcode;             /* signal code to send */
```

Function

`_ss_dcon` sends a signal (sigcode) to the calling process when the “data carrier detect line” associated with the device is present. –1 is returned and errno is set if unable to register the signal.

See Also

`I$SetStt` in the OS-9 Technical Manual.

_ss_dsrts()

Disables RTS Line

Synopsis

```
int _ss_dsrts(path)
int path;                    /* path number */
```

Function

`_ss_dsrts()` disables the RTS line for the device open on path. –1 is returned and errno is set on failure to disable RTS line.

See Also

`I$SetStt` in the OS-9 Technical Manual.

_ss_enrts()

Enables RTS Line

Synopsis

```
int _ss_enrts(path)
int path;                    /* path number */
```

Function

`_ss_enrts()` enables the RTS line for the device open on path. –1 is returned and errno is set on failure to enable the RTS line.

See Also

`I$SetStt` in the OS-9 Technical Manual.

_ss_lock()

Lock Out a Record

Synopsis

```
int _ss_lock(path, locksize)
int path;                  /* path number */
unsigned locksize;         /* number of bytes to lock of file */
                           /* if 0, all locks removed */
                           /* if 0xffffffff, entire file locked out */
```

Function

`_ss_lock()` locks out a section of the file open on path from the current file position up to the number of bytes specified by locksize.

If locksize is zero, all locks (record-lock, EOF lock, and file lock) are removed. If a locksize of 0xffffffff is requested, the entire file is locked out regardless of where the file pointer is. This is a special type of file lock that remains in effect until released by _ss_lock(path,0), a read or write of zero bytes, or the file is closed.

If an error occurs, _ss_lock() returns –1 as its value and the appropriate error code is placed in the global variable errno.

See Also

`I$SetStt` and the section on RBF record-locking in the OS-9 Technical Manual.

_ss_opt()

Set Path Options

Synopsis

```
#include <sgstat.h>

_ss_opt(path, buffer)
int path;                  /* path number */
struct sgbuf *buffer;      /* pointer to buffer for path desc info */
```

Function

`_ss_opt()` copies the buffer pointed to by buffer into the options section of the path descriptor open on path.

Generally, a program gets the options with _gs_opt(), changes the desired values, and updates the path options with _ss_opt(). The structure sgbuf declared in <sgstat.h> provides a convenient means to access the individual option values.

If the path was invalid, _ss_opt() returns –1 and the appropriate error code is left in the variable errno.

Caveats

It is common practice to preserve a copy of the original options so the program can restore them prior to exiting. The option changes take effect on the currently open path (and any paths created with I$Dup to the same).

See Also

`_gs_opt()`; `I$SetStt` in the OS-9 Technical Manual; tmode in Using Professional OS-9.

_ss_pfd()

Put File Descriptor Sector

Synopsis

```
#include <direct.h>

int _ss_pfd(path, buffer)
int path;                  /* path number */
struct fildes *buffer;     /* pointer to buffer for file desc info */
```

Function

`_ss_pfd()` copies certain bytes from the buffer pointed to by buffer into the file descriptor sector of the file open on path. The buffer is usually obtained from _gs_gfd(). Only the owner ID, the modification date, and creation date are changed.

The structure fildes declared in <direct.h> provides a convenient means to access the file descriptor information.

If an error occurs, the function returns the value –1 and the appropriate error value is placed in the global variable errno.

Caveats

The buffer must be at least 32 bytes long or garbage could be written into the file descriptor sector. Only the superuser can change the owner ID field. It is impossible to change the file attributes with this call. Instead, use _ss_attr().

See Also

`_gs_gfd()`; `I$SetStt` and the RBF File Manager in the OS-9 Technical Manual.

_ss_rel()

Release Device

Synopsis

```
int _ss_rel(path)
int path;                  /* path number */
```

Function

`_ss_rel()` cancels the signal to be sent from a device on data ready. `_ss_ssig()` enables a device to send a signal to a process when data is available.

If an error occurs, _ss_rel returns the value –1 and the appropriate error value is placed in the global variable errno.

Caveats

The signal request is also cancelled when the issuing process dies or closes the path to the device. This feature exists only on SCF devices.

See Also

`_ss_ssig()`; `I$SetStt` in the OS-9 Technical Manual.

_ss_rest()

Restore Device

Synopsis

```
int _ss_rest(path)
int path;                    /* path number */
```

Function

_ss_rest() causes an RBF device to restore the disk head to track zero and is usually used for disk formatting and error recovery.

If an error occurs, the function returns the value –1 and the appropriate error value is placed in the global variable errno.

See Also

I$SetStt in the OS-9 Technical Manual.

_ss_size()

Set Current File Size

Synopsis

```
int _ss_size(path, size)

int path;     /* path number */
int size;     /* new size of file in bytes */
```

Function

_ss_size() changes the size of the file open on path. The size change is immediate.

If the size of the file is decreased, the freed sectors are returned to the system. If the size is increased, sectors with undefined contents are added to the file.

If the path is invalid or the device is not a RBF device, a value of –1 is returned as the function value and the appropriate error code is placed in the global variable errno.

Caveats

This call is effective only on RBF devices.

See Also

I$SetStt in the OS-9 Technical Manual.

_ss_ssig()

Send Signal on Data Ready

Synopsis

```
_ss_ssig(path, sigcode)
int path;                    /* path number */
short sigcode;               /* signal code to send */
```

Function

`_ss_ssig()` sets up a signal to send to the calling process when an interactive device has data ready. When data is received on the device indicated by path, the signal sigcode is sent to the calling process.

`_ss_ssig()` must be called each time the signal is sent if it is to be used again.

The device is considered busy and returns an error if any read requests arrive before the signal is sent. Write requests to the device are allowed while in this state.

If an error occurs, the function returns the value –1 and the appropriate error value is placed in the global variable errno.

Caveats

This feature exists only on SCF devices and pipes.

See Also

`_ss_rel()`; `I$SetStt` in the OS-9 Technical Manual.

_ss_tiks()

Wait for Record Release

Synopsis

```
int _ss_tiks(path, tickcnt)
int path;                   /* path number */
int tickcnt;                /* number of ticks to wait for lock */
                            /* if 0, record is released immediately */
                            /* if 1, error returned if not released */
```

Function

If a read or write request is issued for a part of a file that is locked out by another user, RBF normally sleeps indefinitely until the conflict is removed. `_ss_tiks()` may be used to cause an error (E$Lock) to return to the program if the conflict still exists after a specified number of ticks have elapsed.

`tickcnt` specifies the number of ticks to wait if a record-lock conflict occurs with the file open on path. A tick count of zero (RBF's default) causes a sleep until the record is released. A tick count of one means that if the record is not released immediately an error is to be returned.

If an error occurs, _ss_tiks() returns the value –1 and the appropriate error value is placed in the global variable errno.

Caveats

This feature exists only on RBF devices.

See Also

`_ss_rel()`; `I$SetStt` and the section on RBF record-locking in the OS-9 Technical Manual.

_ss_wtrk()

Write Track

Synopsis

```
int _ss_wtrk(path, trkno, siden, ilvf, trkbuf, ilvptr)
int  path;                /* path number */
char *trkbuf,             /* pointer to track buffer image */
     *ilvptr;                          /* pointer to interleave table */
int  trkno,               /* track number to write */
     siden,               /* side of track to write */
     ilvf;                /* interleave factor */
```

Function

`_ss_wtrk()` performs a write-track operation on a disk drive. It is essentially a direct hook into the driver's write-track entry point.

`path` is the path on which the device is open. `trkno` is the desired track number to write. `siden` is the desired side of the track on which to write. `ilvf` is the interleave factor. `trkbuf` is the track buffer image. `ilvptr` is a pointer to an interleave table.

If an error occurs, the _ss_wtrk() function returns the value –1 and the appropriate error value is placed in the global variable errno.

Caveats

This feature exists only on RBF devices. You can obtain additional information on actual use of this call by examining the format utility and/or a device driver.

See Also

`I$SetStt` and RBF device drivers in the OS-9 Technical Manual; format in Using Professional OS-9.

_strass()

Structure Assignment

Synopsis

```
_strass(to, from, count)
char *to,                  /* pointer to copy destination */
     *from;                /* pointer to structure to copy */
int  count;                /* number of bytes to copy */
```

Function

Until the compiler can deal with structure assignment, this function is useful for copying one structure to another. The variable count specifies the number of bytes to copy from memory pointed to by from to memory pointed to by to, regardless of contents.

Caveats

This function can move at most 65536 bytes. No regard is given to overlapping moves.

_sysdate()

Get Current System Date/Time

Synopsis

```
int _sysdate(format, time, date, day, tick)
int   format,             /* date/time format to return */
      *time,              /* pointer to time value */
      *date,              /* pointer to date value */
      *tick;              /* pointer to tick value */
short *day;               /* pointer to day of week value */
```

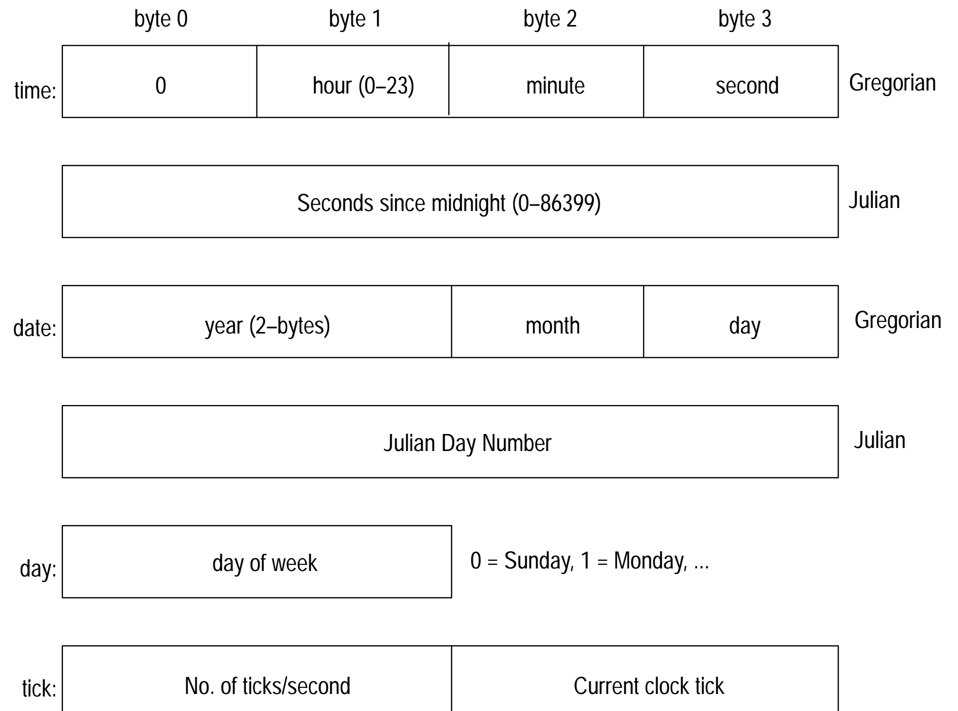
Function

`_sysdate()` obtains the current time, date, day of week, and clock tick from the system. Note that all the arguments except format are pointers to the receiving locations.

format can be any of the following:

0 = Gregorian	2 = Gregorian with ticks
1 = Julian	3 = Julian with ticks

The values are returned in the following format:



If an error occurs, a value of –1 is returned as the function value and the appropriate error code is placed in the global variable errno.

Example

```
main()
{
   int date,time,tick;
   short day;
   .
   .
   .
   _sysdate(0,&time,&date,&day,&tick);
   .
   .
   .
}
```

Caveats

Be careful to pass pointers to the date, time, and day values. Also, be sure day is declared to be a short or the value appears as day + 65536!

See Also

`_julian()`; `F$Julian`, `F$Time` in the OS-9 Technical Manual.

_sysdbg()

Call System Debugger

Synopsis

```
_sysdbg()
```

Function

`_sysdbg()` invokes the system-level debugger, if one exists. This allows you to debug system state routines, such as drivers. The system-level debugger runs in system state and effectively stops timesharing whenever it is active. Only the super user can make this call.

See Also

`F$SysDbg` in the OS-9 Technical Manual.

_tolower()

Convert character to lower case

Synopsis

```
#include <ctype.h>

int _tolower(c)
char c;                    /* character to convert */
```

Function

`_tolower()` is a macro that changes the uppercase argument to lowercase. The argument must be uppercase or garbage results. Use tolower() if the argument is not guaranteed to be uppercase.

See Also

`toupper()`, `isascii()`

_toupper()

Convert Character to Upper Case

Synopsis

```
#include <ctype.h>

int _toupper(c)
char c;                     /* character to convert */
```

Function

`_toupper()` is a macro that changes the lowercase argument to uppercase. The argument must be lowercase or garbage will result. Use toupper() if the argument is not guaranteed to be lowercase.

See Also

`tolower()`, `isascii()`

abs()

Integer Absolute Value

Synopsis

```
int abs(value)
int value;                  /* value to convert */
```

Function

`abs()` returns the absolute value of its integer argument.

Caveats

Applying abs() to the most negative integer yields a result which is the most negative integer:

abs(0x80000000)   returns 0x80000000 as the result.

access()

Determine Accessibility of a File

Synopsis

```
#include <modes.h>

int access(name, perm);
char *name;                /* pointer to name of file */
short perm;                /* file permissions to check */
```

Function

`access()` returns 0 if the mode(s) specified in perm are correct for the user to access name.

The value for perm may be any legal OS-9 mode as defined in the header file <modes.h>. Use a mode value of zero to verify the existence of a file.

If the file cannot be accessed, the function returns –1 and the appropriate error code is placed in the global variable errno.

Caveats

Note that the perm value may not be compatible with other systems.

acos()

Arc Cosine

Synopsis

```
#include <math.h>

double acos(x)
double x;
```

Function

`acos()` returns the arc cosine of x, in the range of 0 to π. The permissible range of x is: $-1 <= x <= 1$.

alm_atdate()

Send a Signal at Gregorian Time and Date

Synopsis

```
int alm_atdate(sigcode, time, date)
int sigcode;     /* signal to be sent to the requester    */
int time;        /* time in the form 00hhmmss (see below) */
int date;        /* date in the form yyyymmdd (see below) */
```

Function

alm_atdate() requests that a signal be sent to the requesting process at a specific Gregorian time and date. The time and date must be in the following format:

	byte 0	byte 1	byte 2	byte 3
time:	0	hour (0–23)	minute	second
date:	year (two bytes)		month	day

Because F$STime may change the system date and time, the alm_atdate() alarm signal is sent when the system time and date become greater than or equal to the alarm time.

If an error occurs, alm_atdate() returns –1 and the appropriate error code is placed in the global variable errno. If no error occurs, alm_atdate() returns the alarm ID.

See Also

F$Alarm in the OS-9 Technical Manual.

alm_atjul()

Send a Signal at Julian Date/Time

Synopsis

```
int alm_atjul(sigcode, time, date)
int sigcode;    /* signal to be sent to the requester  */
int time;       /* number of seconds after midnight    */
int date;       /* Julian day number (see note below!) */
```

Function

`alm_atjul()` requests that a signal be sent to the requesting process at a specific Julian time and date. The date parameter must contain the Julian date on which to send the signal. The Julian date is the number of days since 1 January 4713 B.C., the beginning of the Julian period. The time parameter should contain the time at which to send the signal, expressed as a number of seconds after midnight.

Important: A Julian day begins at midnight on the OS-9 kernel. (Standard Julian days begin twelve hours earlier, at noon.) For example, 1:00 a.m. January 2, 4713 B.C. is one hour after the beginning of Julian Day 1 on the OS-9 kernel, but thirteen hours after the beginning in standard Julian time.

Since F$STime may change the system date and time, the alm_atjul() alarm signal is sent when the system time and date become greater than or equal to the alarm time.

If an error occurs, alm_atjul() returns –1 and the appropriate error code is placed in the global variable errno. If no error occurs, alm_atjul() returns the alarm ID.

See Also

`F$Alarm` in the OS-9 Technical Manual.

alm_cycle()

Send a Signal at Specified Time Intervals

Synopsis

```
int alm_cycle(sigcode, timeinterval)
int sigcode;      /* signal to be sent to the requester */
int timeinterval; /* periodic interval at which the     */
                  /* signal will be sent to the caller  */
```

Function

`alm_cycle()` is similar to the alm_set() function, except that the alarm is reset after it is sent, to provide a recurring periodic signal.

For example, if the request is made at time X, and timeinterval is t, the signal is sent to the requesting process at times (X + t), (X + 2t), (X + 3t), etc. until the alarm request is cancelled.

If the most significant bit of timeinterval is set, timeinterval is assumed to be in 256ths of a second. Otherwise, timeinterval is assumed to be in units of system clock ticks. The minimum timeinterval allowed is two system clock ticks.

If an error occurs, alm_cycle() returns –1 and the appropriate error code is placed in the global variable errno. If no error occurs, alm_cycle() returns the alarm ID.

See Also

`F$Alarm` in the OS-9 Technical Manual.

alm_delete()

Remove a Pending Alarm Request

Synopsis

```
int alm_delete(alarmid)
int alarmid;        /* alarm ID of the request to be cancelled */
```

Function

`alm_delete()` cancels the alarm request specified by alarmid. If zero is passed as the alarm ID, all pending alarm requests are cancelled.

If an error occurs, alm_delete() returns –1 and the appropriate error code is placed in the global variable errno. If no error occurs, alm_delete() returns zero.

See Also

`F$Alarm` in the OS-9 Technical Manual.

alm_set()

Send a Signal After Specified Time Interval

Synopsis

```
int alm_set(sigcode, time)
int sigcode;       /* signal to be sent to the requester      */
int time;          /* interval to pass between when the alarm */
                   /* is requested and when the signal is to  */
                   /* be sent                                 */
```

Function

`alm_set()` sends the signal specified by sigcode to the requesting process after the time specified by time has elapsed. For example, if the request is made at time X and time is t, the signal is sent at time (X + t) unless the alarm request is cancelled.

If the most significant bit of time is set, time is assumed to be in 256ths of a second. Otherwise, time is assumed to be in units of system clock ticks. The minimum time allowed is two system clock ticks.

If an error occurs, alm_set() returns –1 and the appropriate error code is placed in the global variable errno. If no error occurs, alm_set() returns the alarm ID.

See Also

`F$Alarm` in the OS-9 Technical Manual.

asctime()

Convert Broken-Down Time to String Format

Synopsis

```
#include <time.h>

char *asctime(tp)
struct tm *tp              /* pointer to "broken-down" time structure */
```

Function

`asctime()` converts the **Broken-Down** time structure pointed to by tp into the following 26 byte string format (including the terminating \0):

```
xxx mmm dd hh:mm:ss yyyy\n\0
```

xxx is one of the following days of the week:

Sun	Mon	Tue	Wed
Thu	Fri	Sat	

mmm is one of the following months of the year:

Jan	Feb	Mar	Apr
May	Jun	Jul	Aug
Sep	Oct	Nov	Dec

Caveat

`asctime()` returns a pointer to a static area which may be overwritten. To insure data integrity use the string or save it immediately.

See Also

`mktime()` (for the "Broken-Down" Time structure), `ctime()`, `localtime()`.

asin()

Arc Sine

Synopsis

```
#include <math.h>

double asin(x)
double x;
```

Function

`asin()` returns the arc sine of x, in the range $-\pi/2$ to $\pi/2$. The permissible range of x is: $-1 <= x <= 1$.

atan()

Arc Tangent

Synopsis

```
#include <math.h>

double atan(x)
double x;
```

Function

`atan()` returns the arc tangent of x, in the range $-\pi/2$ to $\pi/2$.

atof()

Alpha to Floating Conversion

Synopsis

```
#include <math.h>

double atof(string)
char *string;              /* pointer to string to convert */
```

Function

`atof()` converts the string pointed to by string into its equivalent representation in type double.

`atof()` recognizes the following syntax: an optional sign followed by a digit string (possibly containing a decimal point), an optional e or E, an optional sign, and a digit string (no decimal point):

```
[+/-]digits[.digits] [E/e[+/-]integer]
```

Caveats

Overflow causes unpredictable results. There are no error indications.

atoi()

Alpha to Integer Conversion

Synopsis

```
int atoi(string)
char *string;              /* pointer to string to convert */
```

Function

`atoi()` converts the string pointed to by string into its equivalent representation in type int. `atoi()` recognizes an optional sign followed by a digit string:

```
[+/-]digits
```

atol()

Alpha to Long Conversion

Synopsis

```
long atol(string)
char *string;              /* pointer to string to convert */
```

Function

`atol()` converts the string specified by string into its equivalent representation in type long. `atol()` recognizes an optional sign followed by a digit string:

```
[+/-]digits
```

attach()

Attach to a Device

Synopsis

```
#include <modes.h>

char *attach(name, mode)
char *name;                /* pointer to device name */
short mode;                /* device access mode */
```

Function

`attach()` is useful to make a device *known* to the system or to verify that it is already attached. If the device is not already attached, it is placed in the system's device table, static storage is assigned to the device, and its initialization routine is executed. If the device is already attached, it is not re-initialized.

The device name passed in name is attached with the access mode passed in mode. If successful, the device table address is returned as the value of the function.

If the attach fails, –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

`I$Attach` in the OS-9 Technical Manual.

calloc()

Allocate Storage for Array

Synopsis

```
char *calloc(nel, elsize)
unsigned nel,              /* number of elements in array */
         elsize;           /* size of elements */
```

Function

`calloc()` allocates space for an array. `nel` is the number of elements in the array, and elsize is the size of each element. The allocated memory is cleared to zeroes.

This function calls malloc() to allocate memory. If the allocation is successful, calloc() returns a pointer to the area. If the allocation fails, 0 is returned.

Function

Use extreme care to ensure that only the memory assigned is accessed. To modify addresses immediately above or below the assigned memory is sure to cause unpredictable program results.

Caveats

Use extreme care to ensure that only the memory assigned is accessed. To modify addresses immediately above or below the assigned memory is sure to cause unpredictable program results.

See Also

`malloc()`, `free()`, `ebrk()`, `_srqmem()`, `_srtmem()`

ceil()

Ceiling Function

Synopsis

```
#include <math.h>

double ceil(x)
double x;
```

Function

`ceil()` returns the smallest integer (as a double) that is not less than x.

See Also

`floor()`

chain(), chainc()

Load and Execute a New Module

Synopsis

```
chain(modname, parmsize, parmptr, type, lang, datasize, prior)
char  *modname,            /* pointer to program name */
      *parmptr;            /* ptr to param string to pass to program */
int   parmsize,            /* size of param string */
      datasize;            /* extra memory for program */
short type,                /* module type of program */
      lang,                /* language type of program */
      prior;               /* priority to run program */

chainc(modname, parmsize, parmptr, type, lang, datasize, prior,
       pathent)
char  *modname,            /* pointer to program name */
      *parmptr;            /* ptr to param string to pass to program */
int   parmsize,            /* size of param string */
      datasize;            /* extra memory for program */
short type,                /* module type of program */
      lang,                /* language type of program */
      prior,               /* priority to run program */
      pathent;             /* number of open paths to inherit
```

Function

Use chain() when you need to execute an entirely new program without the overhead of creating a new process. It is functionally equivalent to an os9fork() followed by an exit(), but with less system overhead.

`chain()` effectively **resets** the calling process's program and data areas and begins execution of a new primary module. Open paths are not closed or otherwise affected.

`modname` is a pointer to a null-terminated module name.

`parmptr` is a pointer to a null-terminated string to be passed to the new module. `parmsize` is the strlen() of parmptr.

`datasize` gives extra memory to the new program. If no extra memory is required, this value can be zero.

`type` and `lang` specify the desired type and language of the module; specify these as zero to indicate any type or language.

`prior` is the new priority at which to run the program. Specify zero if no priority change is desired.

If the chain is unsuccessful, chain() returns –1 and the appropriate error code is placed in the global variable errno. If successful, chain() does not return; the new program begins execution.

Regardless of whether the chain is successful, chain() never returns to the caller. Therefore, it is imperative that the caller verifies that the program to chain to exists and is executable before chaining. Use modlink() to check the module directory for the program or modload() to check the execution directory.

`chainc()` is the same as chain() with a pathent argument. pathent is the number of open paths for the new process to inherit. `os9exec()` is the preferred method by which to chain to C programs. See the discussion of os9exec() for more details.

Caveats

Beware of chaining to a system object module. Usually, it only makes sense to chain to a program of type object module. `chain()` is a historical function and is likely to be removed in a future release. Use chainc() instead.

See Also

`os9fork()`; `F$Chain`, `F$Fork` in the OS-9 Technical Manual.

chdir()

Change Current Data Directory

Synopsis

```
chdir(dirname)
char *dirname;              /* pointer to directory pathlist */
```

Function

`chdir()` changes the current data directory for the calling process. The argument dirname is a pointer to a string that gives a pathname for a directory.

`chdir()` returns 0 after a successful call. If dirname is not a directory pathname, a value of –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

This function changes the data directory only for the program containing the function call, not the shell that executes the program. Use the built-in shell command chd to change the shell's data directory.

See Also

`I$ChgDir` in the OS-9 Technical Manual; `chd` in Using Professional OS-9.

chmod()

Change File Access Permissions

Synopsis

```
#include <modes.h>

chmod(name, perm)
char *name;                /* pointer to file name */
short perm;                /* file permissions */
```

Function

`chmod()` changes the access permission bits associated with a file. name must be a pointer to a string containing a file name, and perm should contain the desired bit pattern.

`chmod()` returns 0 after a successful call. If the caller is not entitled to change the access permissions or the file cannot be found, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

Only the super user or the owner of the file may change the access permissions.

See Also

`_ss_attr()`; `I$SetStt` in the OS-9 Technical Manual; attr in Using Professional OS-9.

chown()

Change Owner of a File

Synopsis

```
chown(name, newowner)
char *name;                 /* pointer to file name */
int newowner;               /* new owner group.user ID */
```

Function

`chown()` changes the owner number of a file. name points to the file name to change; the new owner ID is passed in newowner.

The owner ID is made up of a group ID and a user ID. The group ID is passed in the high order **lower byte** and the user ID is passed in the low order lower byte. The owner ID, however, is stored as a word; the group ID in the high byte and the user ID in the low byte. This will change in future revisions of RBF.

`chown()` returns 0 after a successful call. If the caller is not entitled to change the owner ID or the file cannot be found, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

Only the super user may change the owner ID.

See Also

`_ss_pfd()`; `I$SetSt` in the OS-9 Technical Manual.

chxdir()

Change Current Execution Directory

Synopsis

```
chxdir(dirname)
char *dirname;             /* pointer to directory name */
```

Function

`chxdir()` changes the current execution directory for the calling process. dirname is a pointer to a string that gives a pathname for a directory.

`chxdir()` returns 0 after a successful call. If dirname is not a directory pathname, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

This function changes the execution directory only for the program containing the function call, not the shell that executes the program. Use the built-in shell command chx to change the shell's execution directory.

See Also

`I$ChgDir` in the OS-9 Technical Manual; `chx` in Using Professional OS-9.

cleareof()

Clear End of File Condition

Synopsis

```
#include <stdio.h>

cleareof(fp)
FILE *fp;                    /* pointer to file */
```

Function

`clearof()` resets the end-of-file condition that causes feof() to return a non-zero value. This is useful to retry an input operation on a terminal after end-of-file is encountered. The getc() function refuses to read characters until the end-of-file condition is cleared.

Caveats

Be sure to give cleareof() a file pointer and not a path number. `cleareof()` is implemented as a macro in <stdio.h>.

See Also

`clearerr()`, `getc()`

clearerr()

Clear Error Condition

Synopsis

```
#include <stdio.h>

clearerr(fp)
FILE *fp;                  /* pointer to file */
```

Function

`clearerr()` resets the error condition that causes ferror() to return a non-zero value. You can use this to retry an input operation on a terminal after an error is encountered. The getc() function refuses to read characters until the error condition is cleared.

Caveats

`clearerr()` resets the error condition on the file. This does not fix the file or prevent the error condition from occurring again. Be sure to give clearerr() a file pointer and not a path number. `clearerr()` is implemented as a macro in the <stdio.h> header file.

See Also

`cleareof()`, `getc()`

clock()

Get Processor Time

Synopsis

```
#include <time.h>

clock_t  clock()
```

Function

`clock()` returns a value (in **tick** units) approximating the processor time used by the current process. The returned value may be divided by CLK_TCK to determine the processor time in seconds. `CLK_TCK` is defined in time.h to be the number of ticks per second.

If clock() cannot determine the processor time, it returns (clock_t) –1 to indicate an error.

Caveat

`CLK_TCK` is not a syntactic constant. However, it is rarely changed without rebooting the system.

See Also

`F$SetSys`, `F$Time` in the OS-9 Technical Manual.

close()

Close a Path

Synopsis

```
close(path)
int path;                    /* path number */
```

Function

`close()` closes an open path. The path number is usually obtained by a previous call to open(), creat(), create(), or dup(). The standard paths 0, 1, and 2 (standard input, standard output and standard error, respectively) are not normally closed by user programs.

If an error occurs during the close, this function returns –1 and the appropriate error code is placed in the global variable errno.

Caveats

Be sure to use only a path number here, not a file pointer assigned by fopen().

See Also

`open()`, `creat()`, `create()`, `dup()`

closedir()

Closes the Named Directory Stream

Synopsis

```
#include <dir.h>

closedir(dirp)
dir *dirp;                /* pointer to directory stream */
```

Function

`closedir()` closes the named directory stream and frees the structure associated with dirp.

See Also

`opendir()`, `readdir()`, `rewinddir()`, `seekdir()`, `telldir()`

cos()

Cosine

Synopsis

```
#include <math.h>

double cos(x)
double x;
```

Function

`cos()` returns the cosine of x as a double float. The value of x is in radians.

crc()

Calculate Module CRC

Synopsis

```
#include <module.h>

crc(ptr, count, accum)
char     *ptr;           /* ptr to byte to start CRC calculation */
unsigned count;          /* # of bytes over which to calculate CRC */
int      *accum;         /* pointer to CRC accumulator */
```

Function

`crc()` generates an OS-9 CRC (cyclic redundancy check) value for use by compilers, assemblers, or other memory module generators.

The CRC is calculated starting at the byte pointed to by ptr for count bytes. accum is a pointer to the CRC accumulator. It is unnecessary to cover the entire module in one call; the CRC may be accumulated over several calls. The accumulator must be initialized to –1 before the first call.

When verifying the CRC of a module, initialize the accumulator to –1 and perform the CRC over the entire module, including the CRC bytes in the module. If the resulting CRC is equal to the CRC constant value, the module is valid. A manifest constant CRCCON is defined in the <module.h> header file for this use.

To generate the CRC for a module:

- initialize the accumulator to –1
- perform the CRC over the module
- call crc() with a NULL value for ptr
- complement the CRC accumulator
- write the contents of the accumulator to the module

Caveats

The CRC value is three bytes long in a four byte field. To generate a valid module CRC, the caller must include the byte preceding the CRC in the calculation. crc() assumes a zero byte is to be CRCed if ptr is passed as zero.

See Also

`F$CRC` in the OS-9 Technical Manual.

Example

The following code fragment illustrates a technique for generating a CRC for a module:

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

main()
{
  char *ptr;
  int count,accum = -1;

  crc(ptr,count,&accum);  /* one or more calls to crc */

  crc(ptr,count,&accum); /* The final call to crc(). At this point */
                         /* the entire module less the four crc    */
                         /* bytes have been processed.            */

  crc(NULL,0,&accum);    /* this is a special case to run a null   */
                         /* byte through the crc calculation.     */

  accum = ~accum;   /* complement the accumulator. Previous calls  */
                    /* have left the most significant byte as Oxff */
                    /* Complementing changes this byte to zero    */

  fwrite(&accum,1,sizeof accum,fp);
    /* finally write out the four crc bytes  */
}
```

creat()

Create a File

Synopsis

```
#include <modes.h>

int creat(name, mode)
char *name;                /* pointer to file name */
short mode;                /* access mode for file */
```

Function

creat() returns a path number to a new file with the name specified by a string pointed to by name. The file is available for writing. The permissions are given by mode. The owner of the file is the task owner.

If, however, name is the name of an existing file, the file is truncated to zero length, and the ownership and permissions remain unchanged. The file is open for write access.

If an error occurs, the value –1 is returned and the appropriate error code is placed in the global variable errno.

Important: creat() does not return an error if the file exists. Use the access() function to establish the existence of a file if it is important that a file should not be over-written. The valid mode values are available in the <modes.h> header file.

Caveats

It is unnecessary to specify write permissions in mode to write to the file. You cannot create directories with this call. Instead, use mknod().

See Also

open(), create(); I$Create, I$SetStt in the OS-9 Technical Manual.

create()

Create a File

Synopsis

```
#include <modes.h>

int create(name, mode, perm [,initial_size])
char  *name;              /* pointer to file name */
short mode,               /* access permissions */
      perm;               /* file permissions */
[int  initial_size;]      /* initial size of file (optional) */
```

Function

`create()` returns a path number to a new file with a name specified by the string pointed to by name. The file is available for writing. mode gives the access permissions. perm gives the file permission attributes. The owner of the file is the task owner.

If the file already exists or any other error occurs, –1 is returned and the appropriate error code is placed in the global variable errno. The valid mode and permission values are available in the <modes.h> header file.

You may specify the initial_size value to indicate the file's initial allocation size. For disk files, the allocation is made even though the file size does not change. For pipe files, the pipe buffer is set to this value. The initial_size parameter may be used only if the S_ISIZE bit is set in mode.

Caveats

This function is similar to the creat() call except it allows the caller to give the exact file attributes desired and does not truncate the file if it is already present. You cannot create directories with this call. Instead, use mknod().

See Also

`open()`, `creat()`; `I$Create` in the OS-9 Technical Manual.

ctime()

Convert Calendar Time to String Format

Synopsis

```
#include <time.h>

char *ctime(t)
time_t *t;    /* pointer to calendar time struct */
```

Function

`ctime()` converts the Calendar Time t into the following 26 byte string format (including the terminating \0):

```
xxx mmm dd hh:mm:ss yyyy\n\0
```

xxx is one of the following days of the week:

Sun	Mon	Tue	Wed
Thu	Fri	Sat	

mmm is one of the following months of the year:

Jan	Feb	Mar	Apr
May	Jun	Jul	Aug
Sep	Oct	Nov	Dec

`ctime()` is the exact equivalent of:

```
asctime(localtime(t))
```

Caveat

`ctime()` returns a pointer to a static area which may be overwritten; to ensure data integrity use the string or save it immediately.

See Also

`asctime()`, `localtime()`

detach

Detach a Device

Synopsis

```
int detach(ptr)
char *ptr;    /* pointer to device returned by attach() */
```

Function

`detach()` removes a device from the system device table if it is not in use by any other process. If this is the last use of the device, the device driver's termination routine is called. Any permanent storage assigned to the device is deallocated.

`ptr` must be a pointer returned by the attach() call.

If an error occurs, the function returns –1 and the appropriate error code is placed in the global variable errno.

See Also

`attach()`; `I$Detach` in the OS-9 Technical Manual.

difftime()

Find Temporal Difference

Synopsis

```
#include <time.h>

double difftime (t1, t0)
time_t t0,                /* calendar time struct */
       t1;                /* calendar time struct */
```

Function

`difftime()` returns the difference in seconds between t1 and t0. This value is returned as a double. If time_t values could simply be subtracted, the return value would be: t1 – t0.

See Also

`time()`

dup()

Duplicate a path

```
int dup(path)
int path;    /* path number of path to duplicate*/
```

Function

Given an existing path, dup() returns a synonymous path number for the same file or device. The lowest available path number is used.

`dup()` merely increments the link count of a path descriptor and returns a different synonymous path number. The path descriptor is not cloned.

Important: It is usually not a good idea for more than one process to be doing I/O on the same path concurrently. On RBF files, unpredictable results may occur. Use separate I/O paths.

See Also

`I$Dup` in the OS-9 Technical Manual.

ebrk()

Obtain External Memory

Synopsis

```
extern int _memmins;
char *ebrk(size)
unsigned size;            /* number of bytes to return */
```

Function

`ebrk()` returns a specified amount of memory (size). The memory is obtained from the system via the F$SrqMem system request. It is intended for general purpose memory allocation.

The blocks of memory returned by this call may not be contiguous, thereby providing the ability to obtain a block of memory of a given size from anywhere in the 68000 address space.

To reduce the overhead involved in requesting small quantities of memory, ebrk() requests memory from the system in a minimum size determined by the global variable _memmins which is initially set to 8192, and satisfy the user requests from this memory space. ebrk() grants memory requests from this memory space provided the requests are no larger than the amount of space.

If the request is larger than the available space, ebrk() wastes the rest of the space and tries to get enough memory from the system to satisfy the request.

This method works well for programs that need to get large amounts of not necessarily contiguous memory in little bits and cannot afford the overhead of malloc().

Changing the _memmins variable causes ebrk() to use that value as the F$SrqMem memory request size.

If the memory request is granted, a pointer (even-byte aligned) to the block is returned. If the request is not granted, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

The memory obtained from ebrk() is not given back until the process terminates.

See Also

`sbrk()`, `ibrk()`, `malloc()`; `F$SrqMem` in the OS-9 Technical Manual.

exit()

Terminate Task

Synopsis

```
exit(status)
short status;              /* exit status:  if zero, normal exit */
                           /*               else, error code */
```

Function

`exit()` is the normal means of terminating a task. `exit()` does any clean up operations required before terminating, such as flushing out any file buffers.

An exit status of zero is considered normal termination; a non-zero value is interpreted as an error code by most programs (especially the shell).

`exit()` returns no value.

See Also

The discussion of _exit(); the discussion of F$Exit in the OS-9 Technical Manual.

exp()

Exponential Function

Synopsis

```
#include <math.h>

double exp(x)
double x;
```

Function

`exp()` returns the value e (2.71828...) raised to the power x.

fabs()

Floating Absolute Value

Synopsis

```
#include <math.h>

double fabs(x)
double x;
```

Function

`fabs()` returns the absolute value of the argument x.

fclose()

Close a File

Synopsis

```
#include <stdio.h>

int fclose(fp)
FILE *fp;                  /* pointer to file */
```

Function

`fclose()` flushes the buffer associated with file pointer fp. It closes the file and frees the buffer for use by another call to fopen().

It is not considered an error to fclose() a file pointer that is not open, but it is required that fp be obtained from fopen() or be one of the predefined macros stdin, stdout, or stderr.

`fclose()` returns EOF if an error occurs during the close.

See Also

`open()`, `fflush()`, `getc()`, `putc()`

fdopen()

Attach a Path to a File Pointer

Synopsis

```
#include <stdio.h>

FILE *fdopen(path,action)
int  path;                  /* path number */
char *action;               /* pointer to file action string */
```

Function

fdopen() returns a file pointer to the file specified by a currently open path. The action given must be compatible with the access mode of the path given. The valid fdopen() actions are:

Action:	Description:
"r"	Open for reading
"w"	Open for writing
"a"	Append (write) at the end of the file or create file for writing if name does not exist
"r+"	Open for update
"w+"	Create for update
"a+"	Create or open for update at end of file
"d'	Directory read

Use this function when you require some special processing when opening the file that fopen() does not provide.

See Also

fopen(), freopen()

feof()

Check Buffered File for End of File

Synopsis

```
#include <stdio.h>

int feof(fp)
FILE *fp;                    /* pointer to file */
```

Function

`feof()` is a macro used to test a file to see if it is at end-of-file. fp is a pointer to a FILE structure (not a path number).

A non-zero value is returned if the file is at end-of-file; otherwise, a zero is returned.

You can use the macro cleareof() to clear the end-of-file condition.

See Also

`fopen()`, `cleareof()`

ferror()

Check Buffered File for Error Condition

Synopsis

```
#include <stdio.h>

int ferror(fp)
FILE *fp;                    /* pointer to file */
```

Function

`ferror()` is a macro used to test a file to see if an error occurred. fp is a pointer to a FILE structure (not a path number).

A non-zero value is returned if an error occurred from the last I/O operation; otherwise, a zero is returned.

You can use the macro clearerr() to clear the error condition.

Caveats

There is no way to tell what error occurred or when.

See Also

`fopen()`, `clearerr()`

fflush()

Flush a File's Buffer

Synopsis

```
#include <stdio.h>

int fflush(fp)
FILE *fp;                    /* pointer to file */
```

Function

`fflush()` causes a buffer associated with the file pointer fp to be cleared by writing out to the file. The file is written to only if it was opened for write or update.

It is not normally necessary to call fflush(), but it can be useful in the following instances: if the terminal is open on a path other than one of the standard paths and the buffered output needs to be flushed before input; or, a newline character has not yet been issued and you want to deliver the output data to the device.

If a getc() (or equivalent) is performed on stdin, the stdout buffer is flushed automatically.

`fclose()` calls `fflush()` to clean out the buffers before the file is closed.

See Also

`fopen()`, `fflush()`, `getc()`, `putc()`

fgetc()

Get a Character From a File

Synopsis

```
#include <stdio.h>

int fgetc(fp)
FILE *fp;                  /* pointer to file */
```

`fgetc()` returns a character from a file pointed to by fp. EOF is returned on end-of-file or error.

In this implementation, fgetc() is a macro which calls getc(), which is a genuine function. UNIX normally defines getc() as a macro and fgetc() as a genuine function. The usage and result is the same on both systems.

```
getc()
```

fgets()

Get a String From a File

Synopsis

```
#include <stdio.h>

char *fgets(ptr, cnt, fp)
char *ptr;                /* pointer to buffer to hold characters */
int  cnt;                 /* number of characters to read */
FILE *fp;                 /* pointer to file */
```

Function

`fgets()` reads characters from the file fp. It places the characters in the buffer pointed to by the pointer ptr up to an end-of-line character (\n), but not more than cnt-1 characters. A null byte is appended to the end of the string. `fgets()` returns the first argument as its value. `fgets()` returns NULL upon end-of-file.

Caveats

It is the responsibility of the caller to ensure ptr points to a buffer large enough to hold cnt bytes.

See Also

`fgetc()`

fileno()

Determine Path Number From File

Synopsis

```
#include <stdio.h>

int fileno(fp)
FILE *fp;                  /* pointer to file */
```

Function

`fileno()` returns the path number associated with file pointer fp. The path number is not valid unless (fp–>_flag & _INIT) is non-zero.

findnstr()

Search String for Pattern

Synopsis

```
findnstr(pos, string, pattern, len)
int  pos;                  /* position in string to begin search */
char *string,              /* pointer to string to search */
     *pattern;             /* search pattern */
int  len;                  /* length to search */
```

Function

`findnstr()` searches the string pointed to by string for the first instance of the pattern pointed to by pattern. It starts at position pos (where the first position is 1, not zero). The returned value is the position of the first matched character of the pattern in the string, or zero if a match is not found. `findnstr()` stops searching only at position pos + len, so it may continue past null bytes.

Caveats

The current implementation does not use the most efficient algorithm for pattern matching. Use on very long strings is likely to be somewhat slower than it should be.

See Also

`findstr()`

findstr()

Search String for Pattern

Synopsis

```
findstr(pos, string, pattern)
int pos;                    /* position to begin search in string */
char *string,               /* pointer to string to search */
     *pattern;              /* pointer to search pattern */
```

Function

`findstr()` searches the string pointed to by string for the first instance of the pattern pointed to by pattern. It starts at position pos (where the first position is 1, not zero). The returned value is the position of the first matched character of the pattern in the string, or zero if a match is not found. findstr() stops searching when a null byte is found in string.

Caveats

The current implementation does not use the most efficient algorithm for pattern matching. Use on very long strings is likely to be somewhat slower than it should be.

See Also

`findnstr()`

floor()

Floor Function

Synopsis

```
#include <math.h>

double floor(x)
double x;
```

Function

`floor()` returns the largest integer (as a double) that is not greater than x.

fopen()

Open a File

Synopsis

```
#include <stdio.h>

FILE *fopen(name, action)
char *name,                /* pointer to file pathlist*/
     *action;              /* pointer to file action */
```

Function

If the string pointed to by name is a file that can be opened with the action in the string pointed to by action, fopen() returns a pointer to a structure describing a buffered file. If an error occurs, fopen() returns the value 0 (NULL).

The valid fopen() actions are:

Action:	Description:
"r"	Open for reading
"w"	Open for writing
"a"	Append (write) at the end of the file or create file for writing if name does not exist
"r+"	Open for update
"w+"	Create for update
"a+"	Create or open for update at end of file
"d'	Directory read

Important: An x may be appended to any action(s). This indicates that a relative pathlist is relative to the current execution directory. The x also implies that the file should have the execute permission bit set.

Examples

This example creates file for writing in the execution directory.

```
fp = fopen("fred","wx");
```

This example opens existing file for reading.

```
fp = fopen("moe","r");
```

This example creates file for reading and writing.

```
fp = fopen("stooge/curly","w+");
```
C

Caveats

Make sure the argument passed as action is a pointer to a string and not a character; fopen(“fun”,“r”) is correct, fopen(“fun”,’r’) is not.

Opening for a write performs a creat(). If a file with the same name exists when the file is opened for write, it is truncated to zero length. The file is created only if the file does not exist.

Append means open for write and position the file pointer to the end-of-file. This causes the next byte written to be added to the end of the file. If a read is performed, it returns an end-of-file error. Note that the type of a file structure is pre-defined in <stdio.h> as FILE, so a user program may declare or define a file pointer by:

```
FILE *f;
```

Three file pointers are available and are considered open the moment the program runs:

stdin	standard input	I/O is to path 0
stdout	standard output	I/O is to path 1
stderr	standard error	I/O is to path 2

These macros are defined in the header file <stdio.h>.

It is not possible to open a directory for writing with fopen.

See Also

putc(), getc(), creat(), open()

fprint()

Formatted Output

Synopsis

```
#include <stdio.h>

int fprintf(fp, control [,arg0[,arg1...]])
FILE *fp;                  /* pointer to file */
char *control;             /* pointer to control string */
```

Function

`printf()`, `fprintf()`, and `sprintf()` are standard C library functions that perform formatted output. Each of these functions converts, formats, and prints the args (if any) as indicated by the control string.

`fprintf()` places its output on the file pointed to by fp. See the discussion of printf() for details on the control string.

`fprintf()` returns the number of characters output.

Caveats

Use ferror() to check for an output error after calling fprintf().

See Also

`printf()`, `sprintf()`

fputs()

Output a String to a File

Synopsis

```
#include <stdio.h>

fputs(ptr, fp)
char *ptr;                  /* pointer to output string */
FILE *fp;                   /* pointer to file */
```

Function

`fputs()` copies the null terminated string pointed to by ptr into the file fp. The terminating null byte is not copied. –1 is returned if an output error occurs.

See Also

`puts()`, `gets()`, `fgets()`

fread()

Read Data From a File

Synopsis

```
#include <stdio.h>

int fread(ptr, size, nobj, fp)
char *ptr;                  /* pointer to copy buffer */
int  size,                  /* data item size */
     nobj;                  /* number of items to copy */
FILE *fp;                   /* pointer to file */
```

Function

`fread()` reads from the file pointed to by fp. nobj is the number of items to read. Each item is of size bytes. The bytes read are copied into the memory pointed to by ptr. `fread()` returns the number of items read, or zero if an error occurs.

Example

```
func()
{
   int value;
   FILE *file;

   fread(&value, sizeof value, 1, file);
}
```

Caveats

It is the caller's responsibility to ensure enough space is available at ptr.

See Also

`fwrite()`

free()

Return Memory

Synopsis

```
free(ptr)

char *ptr;    /* pointer to memory to be returned */
```

Function

`free()` returns a block of memory granted by malloc() or calloc(). The memory is returned to a pool of memory for later re-use by malloc() or free(). The memory freed by malloc() or free() is returned to the system.

Caveats

It is dangerous to use free() with something other than a pointer previously returned by malloc() or calloc(). To do so hopelessly corrupts the memory lists maintained by malloc(), rendering them useless and possibly causing unpredictable program behavior.

See Also

`malloc()`, `calloc()`

freemem()

Determine Size of Unused Stack Area

Synopsis

```
int freemem()
```

Function

`freemem()` returns the number of bytes allocated for the stack that have not been used.

If compiler stack checking is enabled, the stack is checked for possible overflow before a function is entered. The lowest address the stack pointer has reached is retained so freemem() can report the number of bytes between the stack limit and the lowest stack value as the unused stack memory.

Caveats

The program must be compiled with stack checking code in effect for freemem() to return a correct result. This function is historical; avoid using it in new code as it is likely to be removed in a future release.

See Also

`stacksiz()`

freopen()

Re-Open a File

Synopsis

```
#include <stdio.h>

FILE *freopen(name, action, fp)
char *name,                /* pointer to name of file */
     *action;              /* pointer to action string */
FILE *fp;                  /* pointer to file */
```

Function

`freopen()` is usually used to associate the stdin, stdout, or stderr file pointers with a file instead of a terminal device.

`freopen()` substitutes the file name passed as name instead of the currently open file (if any). The original file is closed with fclose(). The original file is closed even if the opening of the new file does not succeed.

See fopen() for details on the action values.

`freopen()` returns the fp passed or zero if the open failed.

See Also

`fopen()`, `fdopen()`

frexp()

Returns Parts of a Floating Point Number

Synopsis

```
double frexp(value, exp)
double value;              /* floating point value */
int    *exp;               /* exponent */
```

Function

`frexp()` breaks a floating point value into a normalized fraction and an integral exponent of two (exp). This function returns the fraction x, such that 1/2 <= x < 1 and value = x * 2^exp.

fscanf()

Input String Conversion

Synopsis

```
#include <stdio.h>

int fscanf(fp, control [,arg ...])
FILE *fp;                   /* pointer to file */
char *control;              /* pointer to control string */
```

Function

`fscanf()` performs input conversions from data read from the file pointer fp. The format of the control string is described in scanf(). `fscanf()` returns a count of the number of fields successfully matched and assigned.

See the discussion of scanf() for more details.

See Also

`scanf()`, `sscanf()`

fseek()

Reposition File Pointer

Synopsis

```
#include <stdio.h>

int fseek(fp, offset, place)
FILE *fp;                  /* pointer to file */
long offset;               /* file position offset (long same as int) */
int  place;                /* flag to determine offset placement */
```

Function

`fseek()` repositions the file pointer for the buffered file pointed to by fp to a desired character position for the next getc() or putc(). The new position is at offset bytes from:

- the beginning of the file if place is 0
- the current position in the file if place is 1
- the end of the file if place is 2

`fseek()` allows for the special problems of buffering. `fseek()` returns 0 if a seek is reasonable or –1 if the destination (or place) is otherwise incorrect.

Caveats

Using lseek() of a buffered file is certain to cause unpredictable results. fseek() is required when changing from reading to writing or from writing to reading on files open for update.

See Also

`getc()`, `putc()`, `lseek()`

ftell()

Report File Pointer Position

Synopsis

```
#include <stdio.h>

long ftell(fp)
FILE *fp;                    /* pointer to file */
```

Function

`ftell()` is the only proper way to determine the position of the next byte to be read or written when using buffered I/O. `ftell()` returns the current position of the next byte to be read by getc() or written by putc(). The current position is measured in bytes from the beginning of the file. `fp` must be a file pointer.

`-1` is returned if fp does not point to an open file.

Caveats

Do not use lseek() to determine file position as it does not allow for any characters in the buffer.

See Also

`getc()`, `putc()`, `lseek()`

fwrite()

Write Data to a File

Synopsis

```
#include <stdio.h>

int fwrite(ptr, size, nobj, fp)
char *ptr;                  /* pointer to buffer to write */
int  size,                  /* size of items */
     nobj;                  /* number of items to write */
FILE *fp;                   /* pointer to file */
```

Function

`fwrite()` writes to the file pointed to by fp. `nobj` is the number of items to be written. Each item is of size bytes. The bytes written are copied from the memory pointed to by ptr. `fwrite()` returns the number of items written, or zero if an error occurs.

Example

```
func()
{
   int value;
   FILE *file;

   fwrite(&value, sizeofvalue, 1, file);
}
```

See Also

`fread()`

getc(), getchar()

Get Next Character From File, stdin

Synopsis

```
#include <stdio.h>

int getc(fp)
FILE *fp;                  /* pointer to file */
int getchar()
```

Function

getc() returns the next character from the file pointed to by fp. getchar() is a macro that is equivalent to getc(stdin).

The value returned is an int, not a char. This allows the caller to determine the difference between a 0xff byte read from the file and the error condition –1, which is returned when an error or end-of-file occurs.

Under OS-9, there is a choice of low-level service requests to use when reading from a file:

Choice:	Description:
read()	Gets characters up to a specified number of bytes in ***raw*** mode (that is, no editing takes place on the input stream and the characters appear to the program exactly as in the file or on the device).
readln()	Honors the various mapping of characters associated with a Serial Character device such as a terminal, and in any case returns to the caller as soon as a carriage return is seen.

In the vast majority of cases, it is preferable to use readln() for accessing Serial Character devices and read() for any other file input. getc() uses this strategy. As all file input using the Standard Library function is routed through getc(), so do all other library input functions.

The choice is made when the first call to getc() is made after the file is opened. The system is consulted for the status of the file and a flag bit is set in the file structure accordingly.

You can force the choice by setting the relevant bit before a call to getc(). The flag bits are defined in the <stdio.h> header file as _RBF and _SCF.

Use the following method to set the flag bits. Assuming the file pointer (as returned by fopen()) is fp, the following forces the use of readln() and read() on input, respectively:

```
fp->_flag |= _SCF;
fp->_flag |= _RBF;
```

This trick may be played on the standard stream, stdin, before any input is requested from the stream.

When you want to input characters one byte at a time without buffering, use this technique:

```
fp->_flag |= _UNBUF;
```

This causes data to be read from or written to the file one byte at a time.

If a file is open for update (fopen() action r+ or w+), the program is required to do a fseek() before changing from getc() to putc() or putc() to getc(), even if no effective file position change occurs. This causes the buffer to flush (or fill) so input or output can proceed in the opposite direction.

The stdout buffer is flushed automatically by getc(stdin) before the actual read is performed. This ensures that prompts to a terminal are output before the read is issued.

Caveats

It is not a good idea to call read() or write() on a path that is buffered because the buffered data and the low-level I/O may not occur in the correct order.

Important: Due to the buffering, RBF record-locking is ineffective. If an application requires the record-locking facilities, low-level I/O calls and program buffering is required.

See Also

`fopen(), putc()`

getenv()

Value for Environment Name

Synopsis

```
char *getenv(name)
char *name;              /* pointer to environment name string */
```

Function

`getenv()` searches the environment list for a string in the form of name = value. It returns a pointer to the string value if such a string is present, otherwise getenv() returns the value 0 (NULL).

An array of strings called the environment is made available by os9exec() when a process is created. The environment list is usually maintained by the shell. The shell setenv and unsetenv commands can alter the strings. The names of the environment variables and their contents are defined by the application using them.

Important: A process inherits the environment only if the process was created with os9exec(). The os9fork() and chain() functions themselves do not pass the correct information for the argument and environment lists. Do not modify the strings returned by getenv().

See Also

`os9exec()`, `os9fork()`, `chain()`

getime()

Get System Time

Synopsis

```
#include <time.h>

getime(timebuf)
struct sgtbuf *timebuf;    /* ptr to buffer for returned system time */
```

Function

`getime()` returns the system time into the time buffer pointed to by timebuf. The time units are defined in the <time.h> header file.

See Also

`setime()`

getpid()

Determine Process ID Number

Synopsis

```
int getpid()
```

Function

`getpid()` returns the system process ID number for the calling process. You can use this number for such things as creating unique file names.

See Also

`F$ID` in the OS-9 Technical Manual.

gets()

Get a String from a File

Synopsis

```
     #include <stdio.h>

char *gets(ptr)
char *ptr;                 /* pointer to buffer to hold string */
```

Function

`gets()` reads characters from the stdin file and places them in the buffer pointed to by ptr. The function fills the buffer until a carriage return (\n) is read. The \n is replaced by a null byte. `gets()` returns its argument.

Caveats

Because no maximum byte count is available for gets(), it is the caller's responsibility to reserve enough bytes at ptr for the string.

See Also

`fgets()`

getstat()

Get File Status

Synopsis

```
#include <sgstat.h>

getstat(code, path, buffer)
int  code,                  /* code = 0 */
     path;                  /* path number of open file */
char *buffer;               /* pointer to buffer for path options */

getstat(code,path)
int code,                   /* code = 1 or 6 */
    path;                   /* path number of open file */

getstat(code,path,size)
int  code,                  /* code = 2 */
     path;                  /* path number of open file */
long *size;                 /* pointer to current file size */

getstat(code,path,pos)
int  code,                  /* code = 5 */
     path;                  /* path number of open file */
long *pos;                  /* pointer to current file position */
```

Function

`getstat()` is historical from the 6809 C Compiler and is used to access a few I$GetStt system functions. path must be the path number of an open file.

`code` is defined as follows:

Code:	Description:
0	buffer is the address of the 128 byte buffer into which the path option bytes are copied. The <sgstat.h> header file contains a struct defined for use by the program.
1	This applies only to SCF devices and is used to test for data ready. The return value is the number of bytes available or –1 if an error occurred.
2	size is the address of the long integer into which the current file size is placed. The function returns –1 on error and zero on success.
5	pos is the address of the long integer into which the current file position is placed. The function returns –1 on error and zero on success.
6	The function returns –1 on either EOF or error and 0 if not at EOF.

When getstat() returns with the value –1, the appropriate error code is placed in the global variable errno.

Caveats

This function is supported for the convenience of programs ported from the 6809. See the special getstat functions (all of which begin with _gs) for the same information supplied in a more palatable format.

See Also

`setstat()` and all the `_gs` functions (see index)

getuid()

Determine User ID Number

Synopsis

```
int getuid()
```

Function

`getuid()` returns the group/user ID of the current process. The upper word (two bytes) of the value is the group number, the lower word is the user number.

See Also

`F$ID` in the OS-9 Technical Manual.

getw()

Read a Word From a File

Synopsis

```
#include <stdio.h>

int getw(fp)
FILE *fp;                    /* pointer to file */
```

Function

`getw()` returns an integer from the file pointed to by fp. `getw()` returns –1 on error. Therefore, use the macros feof() and ferror() to check the success of getw().

Caveats

`getw()` is a machine-dependent function because the size of a word varies from machine to machine. `getw()` assumes no particular alignment in the file. `getw()` reads two bytes from the file and sign-extends them to four bytes.

gmtime()

Convert Calendar Time to Greenwich Mean Time

Synopsis

```
#include <time.h>

struct *gmtime(tp)
time_t *tp;                /* pointer to Calendar Time structure */
```

Function

`gmtime()` converts the Calendar Time contained in the time structure pointed to by tp to Greenwich Mean Time (GMT).

Caveats

`gmtime()` returns a pointer to a static area which may be overwritten. To ensure data integrity, use the string or save it immediately.

See Also

`mktime()` (for more information on the time structure tp); time().

hypot()

Euclidean Distance Function

Synopsis

```
#include <math.h>

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

Function

`hypot()` returns the Euclidean distance function:

```
sqrt( x * x + y * y)
```

Precautions are taken to avoid unwarranted overflows.

ibrk()

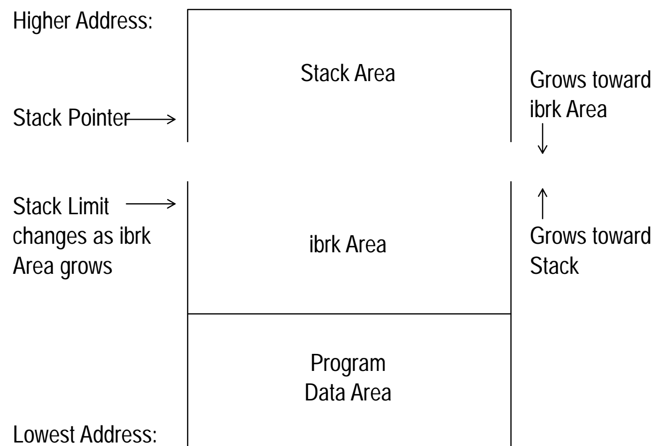
Request Internal mMemory

Synopsis

```
char *ibrk(size)
unsigned size;              /* size of memory block */
```

Function

`ibrk()` returns a pointer to a block of memory of size bytes. The returned pointer is aligned to a word boundary. The memory from which ibrk() grants requests is the area between the end of the data allocation and the stack:



If the requested size would cause the ibrk area to cross the stack pointer, the request fails. You can use freemem() to determine the amount of stack remaining which is also the remaining ibrk area.

`ibrk()` is useful to obtain memory from a fixed amount of memory, unlike ebrk() whose available memory is that of the entire system. The C I/O library functions request the first 2K of I/O buffers from this area, the remainder from ebrk().

Caveats

Be very careful not to crowd out the stack with ibrk() calls. When stack checking is in effect, the program aborts with a `***Stack Overflow***` message if insufficient stack area exists to call a function.

See Also

`sbrk()`, `ebrk()`, `freemem()`, `stacksize()`

index()

Search for Character in String

Synopsis

```
char *index(ptr,ch)
char *ptr,                  /* pointer to string */
     ch;                    /* search character */
```

Function

`index()` returns a pointer to the first occurrence of the character ch in the string pointed to by ptr. If the character is not found, the function returns a NULL (0).

Caveats

This example looks for a period (.) in string and sets ptr to point to it. Note that ch is a character, not a pointer to a character:

```
func()
{
              char *ptr,*string;

              if((ptr = index(string,'.'))
                            process(ptr);
              else printf("No '.' found!\n");
}
```

See Also

`rindex()`

intercept()

Set up Process Signal Handler

Synopsis

```
int intercept(icpthand)
int (*icpthand)();         /* this is how a function declares that */
                           /* an argument is a pointer to a function  */
                           /* returning an int  */
```

Function

`intercept()` instructs OS-9 to pass control to the function whose address is pointed to by icpthand when a signal is received by the process.

If the signal handler function declares an int argument, the function has access to the number of the signal received. On return from the signal handler function, the process resumes at the point in the program where it was interrupted by the signal.

If icpthand is zero, the signal handler is removed.

Caveats

A program will not receive an abort or quit signal from the keyboard (usually ^C and ^E) unless the program performs output to the terminal. This is because OS-9 sends the abort/quit signals to the last process to do I/O to the terminal. If you run the program from the shell and type ^E before the program performs I/O to the terminal, the shell receives the signal and kills the running program. If a program requires control of the terminal immediately, do some I/O to one of the standard paths such as printing a program banner or getting the terminal options with _gs_opt().

Any I/O using the OS-9 C library (for example, printf) cannot be performed inside both the intercept handler function and the main program. Consequently, avoid I/O within intercept functions.

If a math coprocessor (for example, 68881) is present on the system, no floating point math can be used within the intercept handler function.

See Also

`F$Icpt`, `F$Kill`, `F$SigMask` in the OS-9 Technical Manual.

Example

As an example, suppose that a program wants to clean up work files and exit when a keyboard quit signal (signal 3 which, normally, can be caused by typing ^E on the terminal):

```
#include <stdio.h>

#define until(expr)     while(!(expr))

char *tempname = "tempfile";
FILE *tempfp;
int quittime = 0;

/* the signal handler */
gotsignl(signum)
int signum;
{
   switch(signum) {
      case 3:                /* the quit signal */
         quittime = 1;
         break;
      default:               /* ignore all others */
         break;
   }
}

main()
{
    if((tempfp = fopen(tempname,"w") == NULL)
        exit(_errmsg(errno,"can't open file - %s\n",tempname));
    intercept(gotsignl);
    do {
        do_work();
    } until(quittime);
    fclose(tempfp);
    unlink(tempname);
    exit(_errmsg(1,"quittin' time!!!\n"));
}
```

isalnum()

See If Argument is Alphanumeric

Synopsis

```
#include <ctype.h>

int isalnum(c)
char c;
```

Function

`isalnum()` returns a non-zero value if its argument is a digit or an alphabetic character. Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

isalpha()

See If Argument is Alphabetic

Synopsis

```
#include <ctype.h>

int isalpha(c)
char c;
```

Function

`isalpha()` returns a non-zero value if its argument is an alphabetic character. Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

isascii()

See If Argument is ASCII

Synopsis

```
#include <ctype.h>

int isascii(c)
char c;
```

Function

`isascii()` returns a non-zero value if its argument is an ASCII character (that is, the value is less than 128). Otherwise, it returns zero.

See Also

`isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`

iscntrl()

See If Argument is a Control Code

Synopsis

```
#include <ctype.h>

int iscntrl(c)
char c;
```

Function

`iscntrl()` returns a non-zero value if its argument is a control code (values 0 through 31 and 127). Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

isdigit()

See If Argument is a Digit

Synopsis

```
#include <ctype.h>

int isdigit(c)
char c;
```

Function

`isdigit()` returns a non-zero value if its argument is a digit. Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

islower()

See If Argument is Lower Case

Synopsis

```
#include <ctype.h>

int islower(c)
char c;
```

Function

`islower()` returns a non-zero value if its argument is a lower case letter. Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

isprint()

See If Argument is a Printable Character

Synopsis

```
#include <ctype.h>

int isprint(c)
char c;
```

Function

`isprint()` returns a non-zero value if its argument is a printable character (values 32 through 126). Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

ispunct()

See If Argument is a Punctuation Character

Synopsis

```
#include <ctype.h>

int ispunct(c)
char c;
```

Function

`ispunct()` returns a non-zero value if its argument is a punctuation character (neither control nor alphanumeric). Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

isspace()

See If Argument is White Space

Synopsis

```
#include <ctype.h>

int isspace(c)
char c;
```

Function

`isspace()` returns a non-zero value if its argument is one of the white space characters (space, tab, linefeed, return or formfeed). Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

isupper()

See If Argument is Upper Case

Synopsis

```
#include <ctype.h>

int isupper(c)
char c;
```

Function

`isupper()` returns a non-zero value if its argument is an upper case letter. Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

isxdigit()

See If Argument is a Hex Character

Synopsis

```
#include <ctype.h>

int isxdigit(c)
char c;
```

Function

`isxdigit()` returns a non-zero value if its argument is a hexadecimal digit (0-9, A-F, or a-f). Otherwise, it returns zero.

Caveats

The domain of this macro is defined only for c = –1 (EOF) and the values of c that cause isascii() to return a non-zero value.

See Also

`isascii()`

kill

Send a Signal to a Process

Synopsis

```
#include <signal.h>

int kill(pid, sigcode)
int pid;                  /* process ID */
short sigcode;            /* signal code to send */
```

Function

`kill()` sends a signal to a process. Both the sending and receiving process must have the same user number unless the sending process's user number is that of the super user (0).

The value in sigcode is sent as a signal to the process whose ID number is pid. You can give any value in sigcode. The conventional code numbers are defined in the <signal.h> header file.

`kill()` returns –1 if an error occurs and the appropriate error code is placed in the global variable errno.

Caveats

The super user can send a signal to all processes running on the system if the pid is zero.

See Also

`F$Send` in the OS-9 Technical Manual; the `kill` command in Using Professional OS-9.

ldexp()

Multiply Float By Exponent of 2

Synopsis

```
double ldexp(fp, exp)
double fp;                /* floating point value */
int    exp;               /* exponent */
```

Function

`ldexp()` multiplies a floating point value by an integral power of two. This function returns the value equal to fp * 2^exp.

localtime()

Convert Calendar Time to Local Time

Synopsis

```
#include <time.h>

struct tm *localtime(tp)
time_t *tp;                /* pointer to Calendar Time structure */
```

Function

`localtime()` converts the Calendar Time contained in the time structure pointed to by tp to Local Time.

Caveats

`localtime()` returns a pointer to a static area which may be overwritten. To ensure data integrity, use the string or save it immediately.

See Also

`mktime()` (for more information on the time structure tp); `time()`.

log()

Natural Logarithm

Synopsis

```
#include <math.h>

double log(x)
double x;
```

Function

`log()` returns the natural logarithm of x. The value of x must be positive.

log10()

Common Logarithm

Synopsis

```
#include <math.h>

 double log10(x)
 double x;
```

Function

`log10()` returns the common logarithm of x. The value of x must be positive.

longjmp()

Non-Local Goto

Synopsis

```
#include <setjmp.h>

int longjmp(env,val)
jmp_buf env;                /* program environment buffer */
int val;                    /* error status value */
```

Function

setjmp() and longjmp() allow returning program control directly to a higher level function. They are most useful when dealing with errors and signals encountered in a low-level routine.

The goto statement in C is limited in scope to the function in which it appears (that is, the destination of the goto must be in the same function). Control can only be transferred elsewhere by means of the function call, which returns to the caller. In certain abnormal situations it is preferable to start some section of the code again. But this means returning up a ladder of function calls with error indications all the way.

setjmp() marks a point in the program where a subsequent longjmp() can reach. setjmp() places enough information in the buffer passed in env (as defined in <setjmp.h>) for longjmp() to restore the environment to that existing at the associated call to setjmp().

longjmp() is called with the environment buffer as an argument. The caller of longjmp() can use a value val as, perhaps, an error status indicator.

To set up the setjmp() facility, a function calls setjmp() to initialize the buffer, and if the value returned by setjmp() is zero, the program knows that the call was the **first time through**. If the returned value was non-zero, it was a longjmp() returning from some deeper level of the program.

After longjmp() is completed, program execution continues as if the corresponding setjmp() call had just returned the value val. It is imperative that the function calling setjmp() does not return before the call to longjmp().

longjmp() cannot cause setjmp() to return the value zero as that value is returned by the call to setjmp() itself. If longjmp() is invoked with a val of zero, setjmp() returns 1. All automatic variables have values as of the time longjmp() was called.

Caveats

If longjmp() is called before env is initialized by setjmp(), or if the function calling setjmp() has already returned, absolute chaos is guaranteed.

See Also

`setjmp()`

lseek()

Position File Pointer

Synopsis

```
long lseek(path, position, place)
int path;                  /* open path number of file */
long position;             /* new file position offset */
int place;                 /* flag to determine offset placement */
```

Function

`lseek()` repositions the file pointer for the file open on path, to the byte offset given in position. `place` determines from which file position the offset is based:

0	from the beginning of the file
1	from the current position
2	from the end of the file

Seeking to a location beyond end-of-file for a file open for writing and then writing to it creates a **hole** in the file. The hole contains data with no particular value, usually garbage remaining on the disk.

The value returned is the resulting position in the file. If there is an error, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

On OS-9 the file pointer is a full unsigned 32-bits. UNIX file addresses are limited to 31 bits, and an error is returned if the resulting file pointer is negative.

Do not use lseek() on a buffered file since the buffering routines keep track of the file pointer via fseek().

See Also

`fseek()`; `I$Seek` in the OS-9 Technical Manual.

makdir()

Create a Directory

Synopsis

```
#include <modes.h>

makdir(name, mode, perm [,size])
char *name;                /* pointer to name of directory */
short mode,                /* file access mode */
      perm;                /* file attribute permissions */
[int size;]                /* initial size of directory */
```

Function

`makdir()` is used to create a directory file. The pointer name gives the name of the directory. The desired path access mode is given by mode. `perm` is the disk file attribute permissions for the directory. The initial size of the directory is set to size if the S_ISIZE bit is set in mode.

See Also

`I$MakDir` in the OS-9 Technical Manual.

make_module()

Create a Module

Synopsis

```
#include <memory.h>
#include <module.h>

mod_exec *make_module(name, size, attr, perm, typlang, color)
char  *name;              /* ptr to name of module */
int   size;               /* size of module body */
short attr,               /* attributes/revision of module */
      perm,               /* module permissions */
      typlang;            /* type/language of module */
int   color;              /* memory type in which to make module */
```

Function

`make_module()` creates a memory module with the specified name and attributes in the specified memory. `make_module()` is identical to _mkdata_module() except that make_module() allows you to specify the type/language of the module and color of memory in which to make the module. mod_exec is defined in <module.h>.

`size` specifies the desired memory size of the module in bytes. The size value does not include the module header and CRC bytes. `size` is the amount of memory available for actual use. The memory in the module is initially cleared to zeroes.

`attr` is the attribute/revision word, perm is the access permission word, and typlang specifies the type/language of the module.

`color` indicates the specific type of memory in which to load the module. The file memory.h contains definitions of the three types of memory that you can specify:

Type:	Definition:
SYSRAM	System RAM memory
VIDEO1	Video memory for plane A
VIDEO2	Video memory for plane B

If color is zero, no memory type is specified. Consequently, the module is made in whatever memory the system allocates.

`make_module()` returns a pointer to the beginning of the module header. If the module cannot be created, a value of –1 is returned and the appropriate error code is placed in the global variable errno.

Caveat

The name of the module created by make_module() always begins at offset $34 within the module. This implies that program modules, trap handlers, file managers, device drivers, and device descriptors cannot be made conveniently by this call.

See Also

`_mkdata_module C` function; `F$DatMod` service request in the OS-9 Technical Manual.

malloc()

Allocate Memory From an Area

Synopsis

```
char *malloc(size)
unsigned size;            /* size of memory block to allocate */
```

Function

`malloc()` returns a pointer to a block of memory of size bytes. The pointer is suitably aligned for storing any type of data.

`malloc()` maintains an amount of memory called an **arena** from which it grants memory requests. `malloc()` searches its arena for a block of free memory large enough for the request and, in the process, unites adjacent blocks of free space returned by the free() function. If insufficient memory is available in the arena, malloc() calls ebrk() to get more memory from the system.

`malloc()` returns NULL (0) if there is no available memory or if it detects that the arena is corrupted by storing outside the bounds of an assigned block.

Caveats

Use extreme care to ensure that only the memory assigned by malloc() is accessed. Modifying addresses immediately above or below the assigned memory or passing free(), a value not assigned by malloc(), causes unpredictable program results.

See Also

`free()`, `ebrk()`, `ibrk()`, `sbrk()`

memchr()

Memory Search

Synopsis

```
#include <strings.h>

char memchr(s, val, size)
char *s;                    /* pointer to memory to search */
int val;                    /* search value */
unsigned size;              /* size of memory to search */
```

Function

`memchr()` searches a region of memory pointed to by s for the value val. size specifies the size of the region searched. Each byte of the region is compared by forcing val to an unsigned char and comparing the current byte. The search continues until a match is found or the region is exhausted. If a match is found, a pointer to the matching byte is returned. If no match is found, NULL is returned.

memcmp()

Compare Memory

Synopsis

```
#include <strings.h>

memcmp(s1, s2, size)
char *s1,                   /* pointer to memory to compare */
     *s2;                   /* pointer to memory to compare */
unsigned size;              /* size of memory to compare */
```

Function

`memcmp()` makes a byte by byte comparison of two regions of memory pointed to by s1 and s2. `size` specifies the size of each region. memcmp() returns the value of zero if there are no differing bytes. The returned value is positive if the value of the byte in the region pointed to by s1 is greater than the corresponding byte in the region pointed to by s2. The returned value is negative if the reverse is true.

memcpy

Copy Memory

Synopsis

```
#include <strings.h>

memcpy(dest, src, size)
char *dest,                /* pointer to destination memory */
     *src;                 /* pointer to memory to copy */
unsigned size;             /* size of memory to copy */
```

Function

`memcpy()` copies a specific number of bytes from the region beginning at the location pointed to by src to the region beginning at the location pointed to by dest. `size` specifies the number of bytes copied. Regions may overlap with no ill effects.

memmove()

Move Memory

Synopsis

```
#include <strings.h>

memmove(dest, src, size)
char     *dest,            /* pointer to destination memory */
         *src;             /* pointer to memory to copy */
unsigned size;             /* size of memory to copy */
```

Function

`memmove()` copies a specific number of bytes from the region specified by src to the region specified by dest. The number of bytes copied is specified by size. Regions may overlap with no ill effects.

memset()

Fill Memory

Synopsis

```
#include <strings.h>

memset(s, val, size)
void *s;                    /* pointer to memory to fill */
int val;                    /* value with which to fill memory */
unsigned size;              /* size of memory to fill with val */
```

Function

`memset()` fills the region of memory pointed to by s with the value specified by val. size specifies the size of the memory region. Each byte of the region is filled by forcing val to an unsigned char.

mknod()

Create a Directory

Synopsis

```
#include <modes.h>

mknod(name,perm)
char *name;                 /* pointer to name of directory */
short perm;                 /* file attributes to set  */
```

Function

`mknod()` creates a directory file. The pointer name gives the name of the directory, and perm gives the desired access permissions of the directory.

`mknod()` returns zero if the directory was successfully created. If the creation failed, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

`mknod()` does not make UNIX-style special files as there is no such thing on OS-9. This is a historical function and is likely to be removed in a future release. Use makdir() in all new code.

See Also

`makdir()`; `I$MakDir` in the OS-9 Technical Manual.

mktemp()

Create a Unique File Name

Synopsis

```
char *mktemp(name)
char *name;                /* pointer to name of file */
```

Function

You can use mktemp() to ensure that the name of a temporary file is unique in the system and does not clash with any other file name.

A pointer to a template name is passed to the function. The template string should look like a filename with six trailing X's. `mktemp()` replaces the X's with a letter and the current process ID. The letter is chosen so that the resulting name will not conflict with an existing file.

`mktemp()` returns a pointer to the template or NULL if more than 26 mktemp() files exist.

mktime()

Converts Broken-Down Time to Calendar Time

Synopsis

```
#include <time.h>

time_t mktime(tp)
struct tm *tp;             /* pointer to Broken Down Time structure */
```

Function

`mktime()` converts the Local Time values in the Broken-Down Time structure pointed to by tp into a Calendar Time value. Broken-Down Time has the following format:

```
struct tm {
    int tm_sec;    /* seconds after the minute:  [0,59]  */
    int tm_min;    /* minutes after the hour:    [0,59]  */
    int tm_hour;   /* hours after midnight:      [0,23]  */
    int tm_mday;   /* day of the month:          [1,31]  */
    int tm_mon;    /* months after January:      [0,11]  */
    int tm_year;   /* years after 1900                   */
    int tm_wday;   /* days after Sunday:         [0,6]   */
    int tm_yday;   /* days after January 1:      [0,365] */
    int tm_isdst;  /* Daylight Savings Time (DST) flag   */
                   /* 0 = no, 1 = yes, –1 = unknown      */
};
```

`mktime()` ignores the input values of tm_wday and tm_yday. It does not require the other fields of tm to be in the ranges specified above, but instead **normalizes** the fields and sets tm_wday, tm_yday, and tm_isdst. This allows you to easily do temporal arithmetic without having to worry about mixed-base operations.

Example

The following code fragment will correctly iterate over a 24-hour interval no matter where it starts relative to the beginning of the day, month, or year (presuming the range of representable times is not exceeded):

```
struct tm   when;
int         i;

for (i = 0; i < 24; i++) {
   when.tm_hour++;
   mktime(&when);
}
```

Caveat

Returning –1 to indicate an error implies there is a (somewhat obscure) one second interval that appears to be non-representable. The encoding of times for time_t only represents times ranging from 1902 to 2038 (approximately).

See Also

`getenv()`, `sysdate()`, `julian()`

modcload()

Load Module into Colored Memory

Synopsis

```
#include <memory.h>
#include <module.h>

mod_exec *modcload(modname, mode, memtype)
char *modname;            /* path to module                      */
int  mode,                /* access mode with which to open the file */
      memtype;            /* type code of specific memory type       */
```

Function

`modcload()` searches the module directory for a module with the same name as that pointed to by modname and links to it. If the module is not in the module directory, modname is considered a pathlist and all modules in the specified file are loaded. A link is made to the first module loaded from the file and a pointer to it is returned.

`mode` is the mode which opens the file to load. If any access mode is acceptable, zero can be specified for mode.

`memtype` indicates the specific type of memory in which to load the module. The file memory.h contains definitions of the three types of memory that may be specified:

Type:	Definition:
SYSRAM	System RAM memory
VIDEO1	Video memory for plane A
VIDEO2	Video memory for plane B

If memtype is zero, no memory type is specified. Consequently, the module may be loaded in whatever memory the system allocates.

If the load is successful, modcload() returns a pointer to the module. If the load fails, –1 is returned and the appropriate error code is placed in the global variable errno.

modf()

Return Parts of a Real Number

Synopsis

```
#include <math.h>

double modf(value, iptr)
double value,              /* real number */
       *iptr;              /* pointer to integer part of value */
```

Function

`modf()` returns the signed fractional part of value and stores the integral part in the double pointed to by iptr.

modlink()

Link to a Memory Module

Synopsis

```
#include <module.h>

mod_exec *modlink(modname, typelang)
char *modname;             /* pointer to name of module */
short typelang;            /* type/language value of module */
```

Function

`modlink()` searches the module directory for a module with the same name as that pointed to by modname and links to it, provided that typelang matches the respective value of Type/Language in the module. If the module is found, the link count for the module is incremented by one. If any module type or language is desired, specify zero.

The header file <module.h> contains a structure appropriate for accessing the elements of system-defined memory modules.

If the link is successful, modlink returns a pointer to the module. If the link fails, –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

`modload()`; `F$Link` in the OS-9 Technical Manual.

modload()

Load and Link to a Memory Module

Synopsis

```
#include <module.h>

mod_exec *modload(modname, accessmode)
char *modname;              /* pointer to module name */
short accessmode;           /* access mode for module */
```

Function

`modload()` loads all modules in the file specified by the path modname.

A link is made to the first module loaded from the file and a pointer to it is returned. If any module accessmode is acceptable, specify zero for accessmode.

The <module.h> header file contains a structure appropriate for accessing the elements of system-defined memory modules.

If the load is successful, modload() returns a pointer to the module. If the load fails, –1 is returned and the appropriate error code is placed in the global variable errno.

To load a module via the shell path variable list, see modloadp().

See Also

`modlink()`; `F$Load` in the OS-9 Technical Manual.

modloadp()

Load and Link to a Memory Module Using PATH

Synopsis

```
#include <module.h>

mod_exec *modloadp(modname, accessmode, namebuffer)
char *modname;              /* pointer to module name */
short accessmode;           /* access mode for module */
char *namebuffer;           /* pointer to pathlist of loaded module */
```

Function

`modloadp()` is similar to modload() with one exception: the PATH environment variable determines alternate directories to search for the named module. The search procedure in pseudo-code is as follows:

```
if (modload(name, perm) != SUCCESS) {
   if (errno == E_PNNF && PATH is found in the environment) {
      do {
         get next directory from PATH;
            if (modload(path+name) == SUCCESS) {
               return modptr;
            } else if (errno != E_PNNF) return FAILURE;
         } while (more path elements && !found);
      return FAILURE;
   } return FAILURE;
} else return modptr;
```

`PATH` can contain a list of directories to search if the program is not found in the module directory or the execution directory. `PATH` is set to any number of directory pathlists separated by colons:

```
/h0/cmds:/d0/cmds:/n0/droid/h0/special_stuff/cmds
```

`namebuffer` is a pointer to an array that contains the pathlist of the successfully loaded module. A null string is returned if the load fails to find the file. Any errors other than E_PNNF leave the path string used for the load attempt intact. namebuffer can be NULL, if access to the path string is not required.

`modloadp()` returns a pointer to the module if the load is successful. If the load is unsuccessful, –1 if returned and the appropriate error code is placed in errno.

`os9exec()` uses modloadp() to locate the executable file if the fork or chain attempt fails. The action is the same as the shell utility handling of the PATH environment variable.

See Also

`modload()`, `os9exec()`; `shell` in Using Professional OS-9.

munlink()

Unlink From a Module

Synopsis

```
#include <module.h>

munlink(module)
mod_exec *module;          /* pointer to module */
```

Function

`munlink()` informs the system that the module pointed to by module is no longer required by the process. The module's link count is decremented, and the module is removed from memory if the link count reaches zero. module must have been a pointer returned by modload() or modlink().

`munlink()` returns –1 on error. The appropriate error code is placed in the global variable errno.

See Also

`modlink()`, `modload()`; `F$Unlink` in the OS-9 Technical Manual.

munload()

Unload a Module

Synopsis

```
#include <module.h>

int munload(name, type)
char *name;                /* pointer to name of module */
short typelang;            /* module type/language */
```

Function

`munload()` informs the system that the module whose name is pointed to by name with a type/language of typelang is no longer required by the process. The module's link count is decremented and is removed from memory when the link count reaches zero. A typelang of zero means any type.

`munload()` returns –1 on error. The appropriate error code is placed in the global variable errno.

Caveats

This function differs from munlink() in that it unlinks by module name rather than module pointer. Attempting to unlink a module by performing a link to a module by name to determine its address, then unlinking it twice, does not work because the first unlink removes the module from the process's address space.

See Also

`munlink()`, `modlink()`, `modload()`; `F$UnLoad` in the OS-9 Technical Manual.

open()

Open a File

Synopsis

```
#include <modes.h>

int open(name, mode)
char *name;                /* pointer to name of file */
short mode;                /* file access mode */
```

Function

open() opens an existing file with the specified name. Specify the access mode you want in mode. The values for mode are defined in the header file <modes.h>.

open() returns a path number identifying the file when I/O is performed. If the open fails, –1 is returned and the appropriate error code is placed in the global variable errno.

Reads and writes to the file start at the beginning of the file. You can move the file pointer at any time with the lseek() function.

See Also

creat(), create(), read(), write(), dup(), close(); I$Open in the OS-9 Technical Manual.

opendir()

Open the Directory

Synopsis

```
#include <dir.h>

DIR *opendir (filename)
char *filename             /* pointer to directory name */
```

Function

`opendir()` opens the specified directory and associates a directory stream with it. `opendir()` returns a pointer identifying the directory stream in subsequent operations. The pointer NULL is returned if filename cannot be accessed or if it cannot malloc enough memory to hold the entire file descriptor.

See Also

`closedir()`, `readdir()`, `rewinddir()`, `seekdir()`, `telldir()`

os9exec()

OS-9 System Call Processing

Synopsis

```
os9exec(procfunc, modname, argv, envp, stacksize, priority, pathcnt)
int (*procfunc)();         /* ptr to function to create new process */
char *modname,             /* pointer to module name */
     **argv, /* pointer to argument pointer list */
     **envp; /* pointer to environment pointer list */
unsigned stacksize;        /* additional memory to allocate in bytes */
short priority,            /* priority to run process */
      pathcnt;             /* # of initial open paths for new process */
```

Function

`os9exec()` is a high-level fork/chain interface that prepares the argument and environment list before a process is created. The F$Fork system call passes information to a new process as binary data specified by a pointer and size. It is up to the forked process to interpret the data.

`os9fork()` and `chain()` are C-level hooks to the system calls. By themselves, they provide no real argument information. This leaves the interpretation of a raw argument string up to the cstart portion of the C program. Problems can occur when handling quotes and control characters.

`os9exec()` provides more precise control over the argument strings and supplies the environment variables inherited from the parent process. os9exec() closely simulates the UNIX execve() system call.

`procfunc` is a pointer to a function that creates a new process. This function is normally os9fork, chain(), os9forkc, or chainc().

`modname` is a pointer to a string naming the new primary module.

`argv` is the argument pointer list that the new process receives. By convention, argv[0] is the name of the new module and the end of the argv list is marked by a NULL pointer.

`envp` is the environment pointer list that the new process receives. It points to environment variables and the end of the envp list is also marked by a NULL pointer.

`stacksize` is the additional memory (in bytes) to allocate to the new process's stack memory. A value of zero indicates the default stack size the process requires.

`priority` is the priority value at which the new process runs. If the value is zero, the new process is given the priority of the calling process.

`pathcnt` is given only if the procfunc value indicates os9forkc() or chainc(). This value is the number of open paths to pass to the new process. Normally, you specify three for this argument, causing the three standard paths to be passed. A value of zero passes **no** open paths.

If procfunc indicates os9forkc() or os9fork, and the initial attempt to create the child process fails due to E_PNNF (path name not found), os9exec() calls modloadp() with modname to load the module from disk and then attempts the fork again. This action is the same as the shell handling of executable files.

Important: If using os9exec() when procfunc indicates chain() or chainc(), you must provide a full pathlist for the module.

`os9exec()` returns the value of (*procfunc)(). This is the ID of the child process (for os9fork()). –1 is returned on error.

Important: Any program that executes a system command should use the os9exec() interface. The cstart module can handle parameter strings passed by os9fork() and chain(), but no environment is available and the argv pointer list is separated by white space.

Example

The following code fragment is an example call to os9exec(). The argblk array contains pointers to the arguments in the order that the new program is to receive them. The new process receives a copy of the arguments, not the addresses of the arguments. This example passes the current environment by naming the global variable environ.

```
extern int os9forkc();
extern char **environ;

char *argblk[] = {
              "rename",
              "-x",
              "oldname",
              "newname",
              0,
};

main ()
{
               if ((pid = os9exec(os9forkc,argblk[0],argblk,
                  environ,0,0,3)) > 0) wait(0);
              else printf ("can't fork %s",argblk[0]);
}
```

os9fork(), os9forkc()

Create a Process

Synopsis

```
os9fork(modname, parmsize, parmptr, type, lang, datasize, prior)
char *modname,            /* pointer to module name */
     *parmptr;            /* pointer to parameter list */
int parmsize,             /* size of parameter list */
    datasize;             /* additional memory for process in bytes */
short type,               /* type of module */
      lang,               /* module language */
      prior;              /* initial priority of process */

os9forkc(modname, parmsize, parmptr, type, lang, datasize, prior,
         pathcnt)
char *modname,            /* pointer to module name */
     *parmptr;            /* pointer to parameter list */
int parmsize,             /* size of parameter list */
    datasize;             /* additional memory for process in bytes */
short type,               /* type of module */
      lang,               /* module language */
      prior,              /* initial priority of process */
      pathcnt;            /* # of initial open paths for new process */
```

Function

`os9fork()` creates a process that runs concurrently with the calling process. When the new process terminates, its exit status is available to the forking (parent) process. The new process *inherits* the standard paths (path numbers 0, 1, and 2).

`modname` is a pointer to a null-terminated module name. The argument parmptr is a pointer to a null-terminated string to be passed to the new process; parmsize should be set equal to strlen(parmptr)+1. `type` and lang specify the desired type and language of the module; these can be given as zero to indicate any type or language. `datasize` gives extra memory to the new program. If no extra memory is required, this value can be zero. `prior` is the new priority at which to run the program; indicate zero if no priority change is desired.

If the fork is unsuccessful, os9fork() returns –1 and the appropriate error code is placed in the global variable errno. If the fork is successful, the process ID number of the new process is returned.

`os9forkc()` is the same as `os9fork()` with a pathcnt argument. pathcnt is the number of open paths for the new process to inherit.

`os9exec()` is the preferred method to fork C programs. See the discussion of os9exec() for more details.

Caveats

Beware of forking to a system module. It only makes sense to fork to a program of type object module. This is an historical function and is likely to be removed in a future release. Use os9forkc() instead.

See Also

`chain()`, `os9exec()`; `F$Chain`, `F$Fork` in the OS-9 Technical Manual.

pause

Wait for Signal

Synopsis

```
pause()
```

Function

`pause()` may be used to suspend a task until a signal is received. `pause()` always returns –1 unless the signal received caused the process to terminate, in which case pause() never returns.

See Also

`sleep()`, `kill()`; `F$Sleep` in the OS-9 Technical Manual.

pffinit()

Initialize For Float Output (Obsolete)

Synopsis

```
pffinit()
```

Function

`pffinit()` was used on the 6809 to allow printf() to perform float and double conversions. A dummy function named pffinit() exists here for 6809/68000 portability.

Important: `pffinit()` is an historical function and is likely to be removed in a future release.

pflinit()

Initialize for Longs Output (Obsolete)

Synopsis

```
pflinit()
```

Function

`pflinit()` was used on the 6809 to allow printf() to perform long int conversions. A dummy function named pflinit() exists here for 6809/68000 portability.

Important: `pflinit()` is an historical function and is likely to be removed in a future release.

pow()

Power Function

Synopsis

```
#include <math.h>

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

Function

`pow()` returns x raised to the power y. The values of x and y may not both be zero. If x is not positive, y must be an integer.

prerr()

Print Error Message

Synopsis

```
prerr(path, errnum)
int path;                  /* path of err message file; 0 = stnd format */
short errnum;              /* error number */
```

Function

`prerr()` prints an error message on the standard output path. If path is zero, the message corresponding to the error number errnum is printed as ERROR #hhh.lll, with hhh being the high byte of the error number, lll being the lower. If path is non-zero, that path is assumed to be a file containing error message text which is printed along with the error number.

See Also

`F$PErr` in the OS-9 Technical Manual.

printf()

Formatted Output

Synopsis

```
#include <stdio.h>

int printf(control [,arg0[,arg1...]])
char *control;             /* pointer to control string */
```

Function

`printf()`, `fprintf()`, and `sprintf()` are C standard library functions that perform formatted output. Each of these functions converts, formats, and prints the args (if any) as indicated by the control string. printf() places its output on standard output.

The field width and/or precision may be specified by an asterisk (*) instead of a digit string. In these cases, the field width and/or precision is specified by an int argument. These arguments must appear before the argument to be converted. A negative field width argument is interpreted as a '–' flag followed by a positive field width. A negative precision argument is ignored.

The control string determines the format, type, and number of the following arguments expected by the function. If the control string does not match the arguments correctly, the results are unpredictable.

The control string may contain characters to copy directly to the output, and/or format conversion specifications. Each format specification causes the function to take the next successive argument for output conversion.

A format specification consists of a percent (%) character followed by (in this order):

1. An optional hyphen (–) that means the field is left justified.

2. An optional string of digits indicating the field width required. The field is at least this wide and may be wider if the conversion requires it. The field is right justified (padded on the left) unless the hyphen is present (padded on the right). The default padding character is a space. If the digit string starts with a zero, the padding character is 0.

3. An optional period (.) and a digit string (the precision). For floating point arguments this indicates the number of digits to follow the decimal point on conversion. For strings, the maximum number of characters of the string argument to be printed.

4. An optional character l indicating that the following d, x, or o is the specification for a long integer argument. In this compiler, the types long and int are synonymous; the l has no effect. It appears for ease of porting programs to and from other machines.

5. A conversion character indicates the argument type and the desired conversion. The recognized conversion characters are:

Character:	**Definition:**
d,o,x,X	The integer argument is converted to a signed integer, octal, or hexadecimal, respectively. The ranges for these types are: signed integer –2,147,483,648 to 2,147,483,647 octal 0 to 037777777777 hex 0 to 0xffffffff The conversion X prints the alphabetic letters of a hex number in upper case rather than lower case.
u	The integer argument is converted to an unsigned integer in the range 0 to 4,294,967,295.
f	The double argument is converted to [–]nnn.nnn, where the digits after the decimal point are specified as above. If not given, the precision defaults to six digits. If the precision is zero, no decimal point or following digits are printed.
e,E	The double argument is converted to [–]n.nnne({{char177}})nnn. One digit appears before the decimal point. The precision gives the digits following the decimal point. If no precision is given, six digits are used. If the precision is zero, no decimal point appears. The conversion E produces a number using E instead of e preceding the exponent. The exponent always contains three digits. The resulting value is rounded before printing.
g,G	The double argument is converted to style e or f depending on the value resulting from the conversion. e format is used only if the exponent resulting from the conversion is less than –4 or greater than the precision. If no precision is given, six digits are used. Trailing zeroes are removed from the result. A decimal point appears only if followed by a digit. The resulting value is rounded before printing.
c	The argument is printed as a character.
s	The argument is a pointer to a string. Characters from the string are printed up to a null byte, or until the number of characters indicated by the precision have been printed. If the precision is zero or missing, the characters are not counted.
%	(or any unrecognized character) No argument corresponding; the character is printed.

Caveats

Most errors with printf() are caused by the arguments being passed not corresponding in type and number with the control string, which always cause unpredictable results. Also, passing a NULL pointer (0) as the pointer for the %'s conversion is even more unpredictable.

See Also

`fprintf()`, `sprintf()`, and `scanf()`

Examples

```
printf("Value %d(dec), %x(hex),
%o(octal)\n",val,val,val);
```

In this example, s points to control string.

```
printf(s,x,y,z);
```

This example has a 16–character general float field.

```
printf("%-16.16g",fltans);
```

putc(), putchar()

Put Next Character to File, Standard Out

Synopsis

```
#include <stdio.h>

int putc(c, fp)
int c;                      /* character to write */
FILE *fp;                   /* pointer to file */

int putchar(c)
char c;
```

Function

`putc()` writes the character c to the file pointed to by fp. putchar(c) is equivalent to: putc(c,stdout). putc() and putchar() return –1 if unsuccessful; otherwise, they return the character passed.

Use the following low-level functions when reading from a file:

Function:	Definition:
write()	Outputs characters up to a specified byte count in raw mode. No conversion takes place on the output stream and the characters appear in the file (or device) exactly as written.
writeln()	Honors the various mappings of characters associated with a Serial Character device such as a terminal and in any case returns to the caller as soon as a carriage return is written. Normal output conversion consists of possible conversion of carriage return to carriage return/linefeed, expansion of tab to spaces, handling of screen line count pause, etc.

In the vast majority of cases, it is preferable to use writeln() for accessing Serial Character devices and write() for any other file output. `putc()` uses this strategy. As file output is through putc(), so do all the other library output functions.

The choice is made when the first call to putc() is made after the file is opened. The system is consulted for the status of the file. A flag bit is set in the file structure accordingly.

You may force the choice by setting the relevant bit before a call to putc(). The flag bits are defined in the <stdio.h> header file as _RBF and _SCF.

Use the following method to set the flag bits. Assuming that the file pointer (as returned by fopen()) is fp, the following forces writeln() and write() on output, respectively:

```
fp->_flag |= _SCF;
fp->_flag |= _RBF;
```

You can use this method on the standard streams, stdout and stderr, without the need for calling fopen() but you must do it before any output is performed on the stream.

When you want to output characters one byte at a time without buffering, use this technique:

```
fp->_flag |= _UNBUF;
```

This causes data to be written to the file one byte at a time.

If a file is open for update (fopen() action r+ or w+), the program is required to do a fseek() before changing from getc() to putc() or putc() to getc(), even if no effective file position change occurs. This is done to flush (or fill) the buffer so input or output can proceed in the opposite direction.

Caveats

It is not a good idea to call read() or write() on a path that is buffered because the buffered data and the low–level I/O may not occur in the correct order.

Important: Due to the buffering, RBF record–locking is ineffective. If an application requires the OS-9 record–locking facilities, low–level I/O calls and program buffering is required.

See Also

`fopen(), getc()`

puts()

Output a String to a File

Synopsis

```
#include <stdio.h>

puts(ptr)
char *ptr;                 /* pointer to string */
```

Function

`puts()` copies the null terminated string pointed to by ptr onto the file stdout. With the exception that a \n is written out after the string, the effect is exactly that of:

```
fputs(ptr,stdout)
```

The string terminating zero byte is not copied. puts() returns its first argument, or EOF (–1) if an error occurs.

Caveats

The inconsistency of the new line being appended by puts() and not by fputs() is dictated by history and the desire for compatibility.

See Also

`puts()`, `gets()`, `fgets()`

putw()

Put a Word to a File

Synopsis

```
#include <stdio.h>

int putw(w, fp)
int w;                      /* word to write */
FILE *fp;                   /* pointer to file */
```

Function

putw() writes the integer w to the file pointed to by fp. putw() neither assumes nor causes any special alignment in the file. On success, putw() returns the value it has written. Otherwise, it returns –1.

Caveats

putw(), like getw(), is machine-dependent because the size of the integer it outputs varies with the integer size of the machine on which it resides. This compiler defines int values as four byte quantities, but putw() actually outputs two bytes.

qsort()

Quick Sort

Synopsis

```
qsort(base, n, size, compfunc)
char *base;                /* pointer to array */
int  n,                    /* number of items in array */
     size;                 /* size of items in array */
int (*compfunc)();      /* pointer to function returning int */
```

Function

`qsort()` implements the quick-sort algorithm for sorting an arbitrary array of items.

`base` is the address of the array of n items of size size. compfunc is a pointer to a comparison function supplied by the user (or usually just strcmp()). The comparison function is called by qsort() with two pointers to items in the array to compare. It should return an integer which is less than, equal to, or greater than 0, where, respectively, the first item is less than, equal to, or greater than the second.

Caveats

This function is provided as a convenient sorting function for portability purposes. Because qsort() moves entire records while sorting, any non-trivial use of this function is better replaced with a sort and interface of your choice.

read(), readln()

Read Bytes From a Path

Synopsis

```
int read(path, buffer, count)
int path;                  /* path from which to read */
char *buffer;              /* pointer to buffer for read */
unsigned count;            /* minimum size of buffer */

int readln(path, buffer, count)

int path;                  /* path to read */
char *buffer;              /* pointer to read buffer */
unsigned count;            /* minimum size of buffer */
```

Function

`read()` reads bytes from a path. The path number path is an integer which is one of the standard path numbers 0, 1, or 2, or a path number returned from a successful call to open(), creat(), create(), or dup(). `buffer` is a pointer to space with at least count bytes of memory into which read() puts the data read from the path.

It is guaranteed that at most, count bytes are read, but often less are read, either because the path serves a terminal and input stops at the end of a line or the end-of-file has been reached.

`readln()` causes **line-editing** such as echoing to take place and returns once a \n is read in the input or the number of bytes requested has been read. readln() is the preferred call for reading from the terminal.

`read()` essentially does a raw read, that is, the read is performed without translation of characters. The characters are passed to the program as read.

`read()` and `readln()` return the number of bytes actually read (0 indicating the end of the file). If an error occurs, they return –1 and the appropriate error code is placed in the global variable errno.

Caveats

Notice that end-of-file is returned as zero bytes being read, not an error indication.

See Also

`I$Read`, `I$ReadLn` in the OS-9 Technical Manual.

readdir()

Returns a Pointer

Synopsis

```
#include <dir.h>

struct direct *readdir(dirp)
DIR *dirp                   /* pointer to directory */
```

Function

`readdir()` returns a pointer to a structure containing the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid seekdir() operation.

See Also

`closedir()`, `opendir()`, `rewinddir()`, `seekdir()`, and `telldir()`.

realloc()

Resize a Block of Memory

Synopsis

```
char *realloc(oldptr, size)
char *oldptr;              /* old pointer to block of memory */
unsigned size;             /* size of new memory block */
```

Function

`realloc()` re-sizes a block of memory pointed to by oldptr. `oldptr` should be a value returned by a previous malloc(), calloc(), or realloc().

`realloc()` returns a pointer to a new block of memory. The size of this new block is specified by size. The pointer is aligned for storing any type of data.

If the size of the block of memory pointed to by oldptr is smaller than size, the contents of the old block are truncated and placed in the new block. If size is larger, the entirety of the old block's contents begin the new block.

`realloc`(NULL,size) returns the same result as malloc(size).

`realloc()` returns NULL if the requested memory is not available or is size is specified as zero.

See Also

`malloc()`, `calloc()`, `free()`, `ebrk()`, `ibrk()`, `sbrk()`, and `_freemin()`.

rewind()

Return File Pointer to Zero

Synopsis

```
#include <stdio.h>

int rewind(fp)
FILE *fp;                  /* /* file pointer */
```

Function

`rewind()` is equivalent to:

```
fseek(fp,0,0);
```

Zero is returned if the rewind is successful. Otherwise, –1 is returned.

See Also

`fseek()`

rewinddir()

Resets the Position of the Directory Stream

Synopsis

```
#include <dir.h>

rewinddir(dirp)
DIR *dirp;                 /* pointer to directory */
```

Function

`rewinddir()` resets the position of the named directory stream to the beginning of the directory.

See Also

`closedir()`, `opendir()`, `readdir()`, `seekdir()`, and `telldir()`

rindex()

Search for Character in String

Synopsis

```
#include <strings.h>

char *rindex(ptr, ch)
char *ptr;                /* pointer to string to search */
char ch;                  /* search character */
```

Function

`rindex()` returns a pointer to the last occurrence of the character ch in the string pointed to by ptr. If the character is not found, the function returns a NULL (0).

Caveats

The following example code fragment looks for a period (.) in the string string and sets ptr to point to it. The searching is started from the end of the string and progresses to the front of the string. Note that ch is a character, not a pointer to a character:

```
func()
{
              char *ptr,*string;

              if((ptr = rindex(string,'.')) {
                  process(ptr);
              } else {
                 printf("No '.' found!\n");
              }
}
```

See Also

`index()`

sbrk()

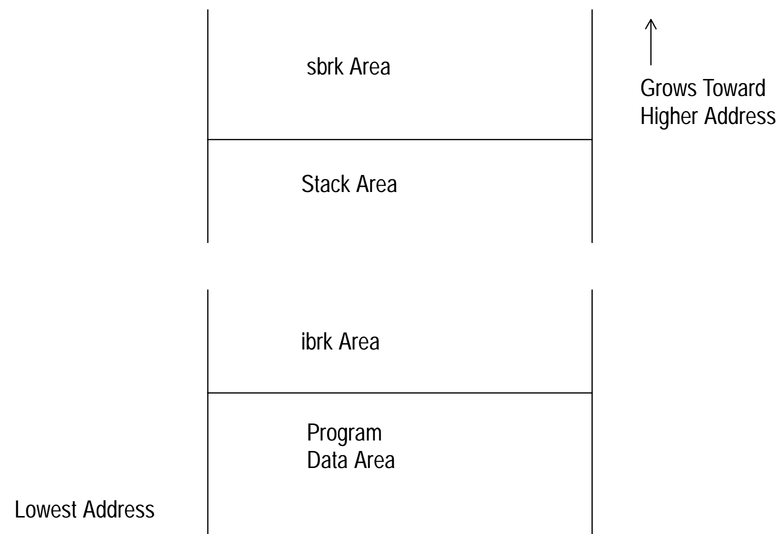
Extend Data Memory Segment

Synopsis

```
char *sbrk(size)
unsigned size;            /* size of memory block desired */
```

Function

`sbrk()` allocates memory from the top of the data area upwards.



`sbrk()` grants memory requests by calling the F$Mem system call. This method resizes the data area to a larger size; the new memory granted is contiguous with the end of the previous data memory.

On systems without an MMU, this call is certain to fail quickly, because it may keep growing in size until the data area reaches other allocated memory. At this point, it is impossible to increase in size and an error is returned. A program may be able to increase its data size only 20K, even if there is 200K available elsewhere.

To gain the most utility of the 68000 addressing space, use the ebrk() function which returns pointers to memory no matter where it is located in the system.

See Also

`ibrk()`, `ebrk()`; `F$Mem` in the OS-9 Technical Manual.

scanf()

Input Strings Conversion

Synopsis

```
#include <stdio.h>

scanf(control [,arg...])
char *control;            /* pointer to control string */
```

Function

`scanf()` performs conversions from the file pointer stdin. This is equivalent to fscan(stdin,control[,arg...]).

Each form of the function (scanf(), fscanf(), and sscanf()) expects a control string, similar to printf, containing conversion specifications, and zero or more pointers to objects into which the converted values are stored.

The functions return EOF (–1) on end of input or error, or a count of the items successfully matched. If the count does not indicate the expected number of items, a match problem occurred.

The control string may contain three types of fields:

- spaces, tab characters, or \n which match any of the three in the input
- characters (not among the above nor %) which must match characters in the input
- a percent sign (%) followed by an optional asterisk (*) denoting suppression of assignment, an optional field width maximum, and a conversion character indicating the type expected

A conversion character controls the conversion to apply to the next field and indicates the type of the corresponding pointer argument. A field consists of consecutive non-space characters and ends when either an input character is inappropriate for the conversion or a specified field width is exhausted. When one field is finished, white space characters are passed over until the next field is found.

The following conversion characters are recognized:

Character:	Definition:
d,o,x	A decimal string, octal string, or hexadecimal string is expected on input, respectively. The argument must be a pointer to an integer.
s	A string of non-space characters is expected and is copied to the buffer pointed to by the corresponding argument with a null byte appended. The caller must ensure the buffer is large enough for the string. The input string is terminated by a space, tab, or newline (\n).
c	A character is expected and is copied into the byte pointed to by the argument. The white space skipping is suppressed for this conversion. If a field width is given, the argument is assumed to point to a character array and the number of characters indicated is copied to it. To ensure that the next non-white space character is read, use %1s with an argument that points to at least four bytes.
e,f	A floating point representation is expected; the argument must be a pointer to a float. Any of the usual ways of writing floating point numbers are recognized.
[	This denotes the start of a set of match characters, the inclusion or exclusion of which delimits the input field. The white space skipping is suppressed. The corresponding argument should be a pointer to a character array. If the first character in the match string is not a circumflex (^), characters are copied from the input as long as they can be found in the match string. If the first character is a circumflex (^), copying continues while the characters cannot be found in the match string. The match string is delimited by a right square bracket (]).
D,O,X	Similar to d,o,x, but the argument is assumed to point to a long. In this compiler, long and int are synonymous.
E,F	Similar to e and f , but the argument is assumed to point to a double.
%	A match for % is sought; no conversion takes place.

The conversion characters d, o, and x may be preceded by l or h to indicate that argument list contains a pointer to a long or short rather than to an int. Each of the functions returns a count of the number of fields successfully scanned. `scanf()` terminates at the end of the control string, when end-of-file is encountered, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Caveats

The returned count of matches/assignments does not include character matches and assignments suppressed by an asterisk (*). The arguments must all be pointers. It is a common error to call scanf() with the value of an item rather than a pointer to it. Also, the \n of an input line must be explicitly matched.

See Also

`fscanf()`, `sscanf()`

seekdir()

Sets the Position of the Next readdir

Synopsis

```
#include <dir.h>

seekdir(dirp, loc)
DIR *dirp;                /* pointer to directory */
long loc;
```

Function

`seekdir()` sets the position of the next readdir() operation on the directory stream. The new position reverts to the one associated with the directory stream when the telldir() operation was performed.

Values returned by telldir() are valid only for the lifetime of the associated dirp pointer. If the directory is closed and then reopened, the telldir() value may be invalidated. It is safe to use a previous telldir() value immediately after a call to opendir() and before any calls to readdir().

See Also

`closedir()`, `opendir()`, `readdir()`, `rewinddir()`, `telldir()`

setbuf()

Fix File Buffer

Synopsis

```
#include <stdio.h>

setbuf(fp, buffer)
FILE *fp;                  /* pointer to file */
char *buffer;              /* pointer to file buffer */
```

Function

Normally, when a file is opened by fopen() and a character is written to or read from it via getc() or putc(), a buffer is obtained from the system (if required) and assigned to the file. `setbuf()` assigns a user buffer to the file instead of the system-assigned buffer. `setbuf()` must be used after the file has been opened and before any I/O has taken place.

The buffer must be of sufficient size and remain in effect until fp is closed. A manifest constant, BUFSIZ, is defined in the <stdio.h> header file that is normally assigned as the buffer size.

If buffer is NULL (0), the file becomes unbuffered and characters are read and written singly.

See Also

`getc()`, `putc()`, `fopen()`

setime()

Set System Time

Synopsis

```
#include <time.h>

setime(timebuf)
struct sgtbuf *timebuf;    /* pointer to time buffer */
```

Function

`setime()` sets the system time from the time buffer pointed to by timebuf. The time units are defined in the <time.h> header file.

If successful, setime() returns zero. Otherwise, –1 is returned and the appropriate error code is placed in the variable errno.

See Also

`getime()`

setjmp()

Non-Local Goto

Synopsis

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;               /* program environment structure */
```

Function

`setjmp()` and `longjmp()` provide a way to perform gotos between functions in C. See the discussion of longjmp() for full details on setjmp().

See Also

`longjmp()`

setpr()

Set Process Priority

Synopsis

```
setpr(pid, prior)
int pid;                    /* process ID */
short prior;                /* new priority for process */
```

Function

`setpr()` sets the process indicated by pid to have a priority of prior. The lowest priority is zero, the highest is 65535.

`setpr()` returns –1 if an error occurs such as the process not having the same user ID as the caller. If an error occurs, the appropriate error code is placed in the global variable errno.

setstat()

Set File Status

Synopsis

```
#include <sgstat.h>

setstat(code, path, buffer)        /* code = 0 */
int code,
    path;                  /* path number */
char *buffer;              /* ptr to buffer containing path desc opts */

setstat(code, path, size)          /* code = 2 */
int code,
    path;                  /* path number */
long size;                 /* new file size */
```

Function

`setstat()` sets the path options or the file size of the file open on path.

If code is zero, the buffer is copied to the path descriptor options section. The header file <sgstat.h> contains the definitions for the path options.

If code is 2, size should be an int specifying the new file size.

If an error occurs, both forms of the call return –1 and place the appropriate error code in the global variable errno.

Caveats

This call exists for 6809 portability. The _ss functions are the preferred versions of these function calls.

See Also

`getstat()`, any C function name beginning with _ss; `I$SetStt` in the OS-9 Technical Manual.

setuid()

Set User ID

Synopsis

```
int setuid(uid)
int uid;                    /* user ID */
```

Function

`setuid()` sets the group/user ID of the process to uid. The following restrictions govern the use of setuid():

- User number 0.0 may change his/her ID to anything without restriction.
- A primary module owned by user 0.0 may change its ID to anything without restriction.
- Any primary module may change its user ID to match the module's owner.

If the call fails, –1 is returned and the appropriate error code is placed in the global variable errno.

See Also

`getuid()`

sigmask()

Controls Process's Signal Handling

Synopsis

```
int sigmask(level)
int level; /* process signal level; 0 = clear                    */
           /*                        1 = increment                */
           /*                       –1 = decrement (not below 0) */
```

Function

Each process descriptor contains an internal variable called the signal mask, which determines the process's signal handling. sigmask() controls the process's signal mask.

If a signal is received by a process whose signal mask is zero, normal program flow is interrupted and the signal is processed by the execution of the program's intercept routine.

If a signal is received by a process whose signal mask is non-zero, the signal is placed in a queue of signals waiting to be processed. The queued signals become active only when the process's signal mask becomes zero.

The process's signal mask is automatically incremented during the execution of its intercept routine. This prevents the intercept routine from being accidentally re-entered if a new signal arrives. The process may use sigmask() within its intercept routine to allow re-entrant signals or to force the signal mask to remain non-zero when normal program execution resumes.

When a process makes an F$Sleep or F$Wait system call, its signal mask is automatically cleared. If any signals are pending, the process returns to the intercept routine without sleeping.

The S$Kill and S$Wake signals ignore the state of the signal mask and are never queued. S$Kill terminates the receiving process, and S$Wake ensures that the receiving process is active.

If an error occurs, sigmask() returns –1 and the appropriate error code is placed in the global variable errno. If no error occurs, sigmask() returns zero.

Caveats

I/O operations using the cio library should not be performed by both the main program and the intercept routine.

If an intercept routine is exited with the longjump() function, the signal mask is still set to one. Generally, the destination of the longjump should unmask signals immediately.

The depth to which signals may queue is limited only by available memory.

See Also

`kill()` and `intercept()` C functions; `F$SigMask` in the OS-9 Technical Manual.

sin()

Sine Function

Synopsis

```
#include <math.h>

double sin(x)
double x;
```

Function

`sin()` returns the sine of x as a double float. The value of x is in radians.

sleep()

Suspend Execution For a Time

Synopsis

```
sleep(seconds)
unsigned seconds;
```

Function

`sleep()` suspends the calling process for the specified time. A sleep time of zero seconds sleeps indefinitely. `sleep()` returns the number of ticks remaining to sleep if awakened prematurely by a signal.

See Also

`tsleep()`; `F$Sleep` in the OS-9 Technical Manual.

sprintf()

Formatted Output

Synopsis

```
#include <stdio.h>

int sprintf(buffer, control [,arg0[,arg1...]])
char *buffer;              /* pointer to output buffer array */
char *control;             /* pointer to control string */
```

Function

`printf()`, `fprintf()`, and `sprintf()` are C standard library functions that perform formatted output. Each of these functions converts, formats, and prints the args (if any) as indicated by the control string.

`sprintf()` places its output into the array pointed to by buffer; the string is terminated by a null byte. The function returns the number of characters placed in the buffer, not including the null byte.

The control string determines the format, type, and number of the following arguments expected by the function. If the control string does not match the arguments correctly, the results are unpredictable. See the discussion of printf() for details on the control string.

The field width and/or precision may be specified by an asterisk (*) instead of a digit string. In these cases, the field width and/or precision is specified by an int argument. These arguments must appear before the argument to be converted. A negative field width argument is interpreted as a '–' flag followed by a positive field width. A negative precision argument is ignored.

See Also

`printf()`, `fprintf()`

sqrt()

Square Root Function

Synopsis

```
#include <math.h>

double sqrt(x)
double x;
```

Function

`sqrt()` returns the square root of x. x must not be negative.

srqcmem()

Allocate Colored Memory

Synopsis

```
#include <memory.h>

char *srqcmem(bytecnt, memtype)
int bytecnt,   /* size of memory to allocate */
    memtype;   /* type of memory to allocate */
```

Function

`srqcmem()` is a direct hook to the F$SRqCMem system call. bytecnt is rounded to a system-defined block size. The size of the allocated block is stored in the global integer variable _srqcsiz. If bytecnt is 0xffffffff, the largest contiguous block of free memory in the system is allocated.

memtype indicates the specific type of memory to allocate. <memory.h> contains definitions of the three types of memory that you may specify:

Type:	Definition:
SYSRAM	System RAM memory
VIDEO1	Video memory for plane A
VIDEO2	Video memory for plane B

If memtype is zero, no memory type is specified. Consequently, any available system memory may be allocated.

If successful, a pointer to the memory granted is returned. The pointer returned always begins on an even byte boundary. If the request was not granted, the function returns the value –1 and the appropriate error code is placed in the global variable errno.

Important: `srqcmem()` is identical to _srqmem() with the exception of the additional color parameter.

See Also

`_srqmem()`, `sbrk()`, `ibrk()`, `ebrk()`, `_srtmem()`, `malloc()`, `free()`; `F$SRqCMem` in the OS-9 Technical Manual.

sscanf()

Input Strings Conversion

Synopsis

```
#include <stdio.h>

sscanf(string, control [,arg...])
char *string;              /* pointer to input string */
char *control;             /* pointer to control string */
```

Function

`sscanf()` performs conversions from the input string pointed to by string. `sscanf()` expects a control string, similar to printf(), containing conversion specifications, and zero or more pointers to objects into which the converted values are stored.

The function returns EOF (–1) on end of input or error, or a count of the items successfully matched. If the count does not indicate the expected number of items, a match problem occurred.

The control string is described in detail in the discussion of scanf().

See Also

`scanf()`, `fscanf()`, `printf()`

stacksiz()

Obtain Size of Stack Used

Synopsis

```
int stacksiz()
```

Function

If the stack checking code is in effect, a call to stacksiz() returns the maximum number of bytes of stack used at the time of the call. You can use this function to determine the stack size a program requires.

Important: This function is historical and will likely be removed in a future release.

See Also

freemem()

strcat()

String Catenation

Synopsis

```
#include <strings.h>

char *strcat(s1, s2)
char *s1,                  /* pointer to original string */
     *s2;                  /* pointer to string to append to s1 */
```

Function

strcat() appends a copy of the string pointed to by s2 to the end of the string pointed to by s1. Null-byte terminated strings are assumed. strcat() returns its first argument.

Caveats

The calling routine is responsible for ensuring adequate space to append s2 to s1.

strcmp()

String Comparison

Synopsis

```
#include <strings.h>

int strcmp(s1, s2)
char *s1,                 /* pointer to string to compare */
     *s2;                 /* pointer to string to compare */
```

Function

`strcmp()` compares the strings pointed to by s1 and s2 for lexicographic order and returns an integer less than, equal to, or greater than zero, where respectively, s1 is less than, equal to, or greater than s2. Null-byte terminated strings are assumed.

strcpy

String Copy

Synopsis

```
#include <strings.h>

char *strcpy(s1, s2)
char *s1,                 /* pointer to destination string */
     *s2;                 /* pointer to source string to copy */
```

Function

`strcpy()` copies characters from s2 to the space pointed to by s1. Null-byte terminated strings are assumed. If s2 is too long, s1 is not null-terminated.

`strcpy()` returns its first argument.

Caveats

`strcpy()` assumes that there is adequate space pointed to by s1 to do the copy. The calling routine is responsible for ensuring that the space is adequate.

strhcpy()

Copy Old OS-9 Strings

Synopsis

```
#include <strings.h>

char *strhcpy(s1, s2)
char *s1,                  /* pointer to destination string */
     *s2;                  /* pointer to source string to copy */
```

Function

`strhcpy()` makes a copy of the string pointed to by s2 in the string pointed to by s1. The s2 string is assumed to have the high-order bit set on the character byte indicating the last character. The function copies the bytes from s2 until the terminator to s1 removes the high-bit from the terminator character. It then appends a null byte to the output string.

`strhcpy()` is used primarily for copying directory names from RBF disks.

The function returns its first argument.

Caveats

`strhcpy()` assumes there is adequate space pointed to by s1 to do the copy. The calling routine is responsible for verifying the space.

strlen()

Determine String Length

Synopsis

```
#include <strings.h>

int strlen(s)
char *s;                   /* pointer to string */
```

Function

`strlen()` returns the number of non-null characters in the string pointed to by s. The function stops when the first null byte is encountered.

strncat()

String Catenation

Synopsis

```
#include <strings.h>

char *strncat(s1, s2, count)
char *s1,                  /* pointer to original string */
     *s2;                  /* pointer to string to append to s1 */
int count;                 /* maximum number of characters to append */
```

Function

`strncat()` appends a copy of the string pointed to by s2 to the end of the string pointed to by s1. The function copies, at most, count characters. Null-byte terminated strings are assumed. strncat() returns its first argument.

Caveats

The function assumes there is room to copy s2 at the end of s1. Space needs to be properly allocated by the calling routine.

strncmp()

String Comparison

Synopsis

```
#include <strings.h>

int strncmp(s1, s2, count)
char *s1,                  /* pointer to string to compare */
     *s2;                  /* pointer to string to compare */
int count;                 /* maximum number of characters to compare */
```

Function

`strncmp()` compares the strings pointed to by s1 and s2 for lexicographic order and returns an integer less than, equal to, or greater than zero, where, respectively, s1 is less than, equal to, or greater than s2. The function compares, at most, count characters. Null-byte terminated strings are assumed.

strncpy()

String Copy

Synopsis

```
#include <strings.h>

char *strncpy(s1, s2, count)
char *s1,                 /* pointer to destination string */
     *s2;                 /* pointer to source string to copy */
int count;                /* maximum number of characters to copy */
```

Function

`strncpy()` copies characters from s2 to the space pointed to by s1, the number of which is determined by count.

If the string s2 is too short, s1 is padded with null bytes to make up the length difference. If s2 is too long, s1 is not null-terminated.

Null-byte terminated strings are assumed. strncpy() returns its first argument.

Caveats

The function assumes that there is adequate space pointed to by s1 to do the copy. The calling routine is responsible for ensuring that the space is adequate.

strtod()

String to Double Conversion

Synopsis

```
#include <math.h>

double strtod(nptr, endptr)
char   *nptr;              /* pointer to beginning of string */
char   **endptr;           /* specifies first character after converted string */
```

Function

strtod() parses a character string and converts it to the associated numeric value of type double. nptr pointes to the beginning of the string. strtod() ignores all leading white space.

If successful, strtod() returns the converted value. The address of the first character past the converted string is pointed to by endptr, if endptr != (char **) NULL.

If the string to be converted is empty, or the string does not have the appropriate form, strtod() returns zero (NULL) and stores nptr in *endptr.

If the converted value causes an overflow, strtod() returns HUGE_VAL, with the appropriate preceding sign. If the converted value causes an underflow, strtod() returns zero. In either of these two cases, ERANGE is placed in errno.

See Also

strtol(), strtoul()

strtol(), strtoul()

String to Long Conversion

Synopsis

```
#include <limits.h>

unsigned long strtol(nptr, endptr, base)
char *nptr;               /* pointer to beginning of string */
char **endptr;            /* specifies first character after converted string */
int  base;                /* base of number string */

unsigned long strtoul(nptr, endptr, base)
char *nptr;               /* pointer to beginning of string */
char **endptr;            /* specifies first character after converted string */
int  base;                /* base of number string */
```

Function

`strtol()` parses a character string and converts it to the associated numeric value of type long. strtoul() converts the string into an unsigned long value. In all other respects, strtoul() and strtol() are identical.

`nptr` points to the beginning of the string. strtol() ignores all leading white space.

`base` is the base of the number string to be converted. For example, strtol(“377”, NULL, 8) returns 255. `base` must be zero or in the range of 2 – 36. If base is 16, strtol accepts a leading 0x or 0X. Digits from 10 through 35 should be represented by the letters ‘a’ through ‘z’ respectively (case is not significant). Only digits valid with the specified base are parsed.

If a base of zero is specified, strtol() accepts any form of constant that the C compiler accepts, excluding l or L, which as a suffix indicates a long constant. Optional preceding signs are also accepted. For example, strtol(“0xffff”, NULL, 0) returns 65535.

If the string to be converted is empty or does not have the appropriate form, strtol() returns zero (NULL) and *endptr contains nptr.

If the converted value causes an overflow, strtol() returns LONG_MAX or LONG_MIN to indicate positive or negative values, respectively. If the converted value causes an underflow, strtol() returns zero. In either of these cases, ERANGE is placed in errno.

See Also

`strtod()`

system()

Shell Command Execution

Synopsis

```
system(string)
char *string;              /* pointer to command line string */
```

Function

`system()` passes its string to the host environment to be executed by the command processor indicated by the SHELL environment variable. If the SHELL variable is not set, the shell command processor is assumed. The command processor executes the argument string as a command line.

The calling process is suspended until the shell command is completed. system() returns the exit status of the created shell.

A null pointer may be used for string to inquire whether the command processor exists. In this case, system() returns a non-zero value if and only if the command processor is available. If passed a non-null pointer, system() returns the exit status of the command processor.

See Also

`os9exec()`, `wait()`

tan()

Tangent Function

Synopsis

```
#include <math.h>

double tan(x)
double x;
```

Function

`tan()` returns the tangent of x as a double float. The value of x is in radians.

Caveats

`tan()` may cause a trapv exception for values close to PI/2.

telldir()

Returns the Current Location

Synopsis

```
#include <dir.h>

long telldir(dirp)
DIR *dirp                  /* pointer to directory */
```

Function

`telldir()` returns the current location associated with the named directory stream.

See Also

`closedir()`, `opendir()`, `readdir()`, `rewinddir()`, `seekdir()`.

tgetent()

Get termcap Entries

Synopsis

```
#include <termcap.h>

extern char *BC;
extern char *UP;
extern char PC_;
extern short ospeed;

int tgetent(bufptr, name)
char *bufptr,              /* ptr to buffer for termcap entry data */
     *name;                /* ptr to name of terminal entry */
```

Function

`tgetent()` extracts the entry for the terminal specified by name. The entry is placed into the buffer pointed to by bufptr. The size of this buffer must be at least 1024 characters and must remain intact for all subsequent calls to tgetnum(), tgetflag(), and tgetstr().

If the termcap file cannot be opened, tgetent() returns –1. If the terminal name cannot be found, 0 is returned. 1 is returned and the data placed in the buffer if the terminal name is found.

The behavior of tgetent() can be modified if the shell environment variables termcap and term are defined. If termcap is found in the environment and its value string begins with a slash (/), that string is used as the path to the file to be used instead the default termcap file. If the value string does not begin with a slash and name is the same as the environment string for term, tgetent() uses this file to search for the terminal entry. This mode is useful for testing or custom termcap definitions. If the termcap value string does not begin with a slash, the string is used as the termcap string instead of reading a file.

Important: `tgetent()` must be called before any of the other termcap library functions.

tgetflag()

Check Terminal Capability Presence

Synopsis

```
#include <termcap.h>

extern char *BC;
extern char *UP;
extern char PC_;
extern short ospeed;

int tgetflag(id)
char *id;                  /* capability ID */
```

Function

`tgetflag()` returns 1 if the capability ID was specified for the terminal and 0 if not specified for the terminal.

tgetnum()

Get Terminal Capability ID

Synopsis

```
#include <termcap.h>

extern char *BC;
extern char *UP;
extern char PC_;
extern short ospeed;

int tgetnum(id)
char *id;                  /* capability ID */
```

Function

`tgetnum()` returns the numeric value given for the capability id. If id was not specified for the terminal, –1 is returned.

tgetstr()

Get Terminal Capability

Synopsis

```
#include <termcap.h>

extern char *BC;
extern char *UP;
extern char PC_;
extern short ospeed;

char *tgetstr(id, strptr)
char *id,                  /* capability ID */
     **strptr;             /* ptr to buffer for capability string */
```

Function

`tgetstr()` places the string given for capability id into the buffer pointed to by the strptr pointer. `strptr` is then advanced past the last character of the returned data.

The special escape codes are given earlier in this chapter. Cursor motion and padding information are interpreted when the string is actually output with tgoto() and tputs().

tgoto

Get Cursor Movement Capability

Synopsis

```
#include <termcap.h>

extern char *BC;
extern char *UP;
extern char PC_;
extern short ospeed;

char *tgoto(motion_string, column, line)
char *motion_string;       /* cm capability string */
int  column,               /* destination column position */
     line;                 /* destination line number */
```

Function

`tgoto()` returns a string suitable for positioning the cursor on the terminal. motion_string is the string given by the cm capability. column and line specify the desired column and line destination for the cursor. The extern variables UP and BC (set to the up and bc capability strings, respectively) are used to avoid placing a null or newline character into the output string.

`tgoto()` returns a pointer to the translated motion string if the motion string was successfully created, otherwise, the string “OOPS” is returned.

time()

Get Calendar Time

Synopsis

```
#include <time.h>

time_t time(timer)
time_t *timer;            /* pointer to returned calendar time */
```

Function

`time()` returns a value approximating the current Calendar Time. If timer is non-null, the return value is stored at the location pointed to by timer. If the Calendar Time is not available, time() returns (time_t) –1.

`time()` looks at the environment variable TZ to determine the local time zone. If TZ is not set, time() uses USA Central Standard Time (CST). TZ should have the following format:

```
zzz[+/-n][:ddd]
```

`zzz` is one of the supported time zone names listed below:

Value:	Description:
GMT, UTC	Greenwich Mean Time a.k.a. Coordinated Universal Time
PST, PDT	USA Pacific Standard Time/Daylight Savings Time
CST, CDT	USA Central Standard Time/Daylight Savings Time
EST, EDT	USA Eastern Standard Time/Daylight Savings Time
YST	Yukon Standard Time (Most of Alaska)
AST	Aleutian/Hawaiian Standard Time
EET	Eastern European Time
CET	Central European Time
WET	Western European Time

`n` is the optional number of minutes east (+) or west (–) of the time zone.

`ddd` is the optional handling Daylight Savings Time. Recognized values for ddd are:

Value:	Description:
no	Do not use Daylight Savings Time.
usa	Conforms to the US Uniform Time Act of 1967 and its various amendments through 1987.
eur	Observe European Daylight Savings Time.

The American time zones specified above default to usa, with the exception of AST which defaults to no. The European time zones specified above default to eur and GMT/UTC defaults to no.

Caveat

Returning –1 to indicate an error, implies there is a (somewhat obscure) one second interval that appears to be non-representable. The encoding of times for time_t only represents times ranging from 1902 to 2038 (approximately).

See Also

`getenv()`, `sysdate()`, and `julian()`.

toascii()

Character Translation

Synopsis

```
#include <ctype.h>

int toascii(c)
int c;
```

Function

`toascii()` returns its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

See Also

`getc()`

tolower()

Character Translation

Synopsis

```
#include <ctype.h>

int tolower(c)
int c;
```

Function

`tolower()` is a genuine function (as opposed to _tolower() which is a macro) that requires a character in the range of –1 to 255. If the argument represents an uppercase letter, the value returned is the corresponding lowercase letter. All other values in the domain are returned unchanged.

See Also

`_tolower()`

toupper()

Charater Translation

Synopsis

```
#include <ctype.h>

int toupper(c)
int c;
```

Function

`toupper()` is a genuine function (as opposed to _toupper() which is a macro) that requires a character in the range of –1 to 255. If the argument represents a lowercase letter, the value returned is the corresponding uppercase letter. All other values in the domain are returned unchanged.

See Also

`_toupper()`

tputs()

Output Capability String

Synopsis

```
#include <termcap.h>

extern char *BC;
extern char *UP;
extern char PC_;
extern short ospeed;

tputs(str, lines_affected, outfunc)
char *str;                 /* pointer to capability string */
int  lines_affected,       /* number of lines affected by operation */
     (*outfunc)();         /* pointer to output function */
```

Function

`tputs()` is used to output the capability strings to the terminal. `tputs()` decodes leading padding information from the str string.

`lines_affected` is the number of lines affected by the operation. `lines_affected` should be set to 1 if not applicable.

`outfunc` is called by tputs() to output each successive character of the str string.

`ospeed` should contain the baud rate code as returned in _sgs_baud by the _gs_opt() function. ospeed is used to calculate the number of fill characters required when fill character delays are required. If ospeed is outside the range of 1–16, no delays are attempted.

The PC_ variable should contain the pad character if it is other than NULL (0).

tsleep()

Sleep for Specified Interval

Synopsis

```
tsleep(svalue)
unsigned svalue;           /* sleep interval:  0 = sleep indefinitely */
                           /*                  1 = one time slice     */
                           /* high bit set: svalue = # of 1/256 secs  */
```

Function

`tsleep()` deactivates the calling process for a specified interval given by svalue. If svalue is zero, the process sleeps indefinitely. If svalue is one, the process gives up a time slice. svalue is considered a tick count to sleep. If the high bit of svalue is set, the remaining 31 bits are considered the number of 256ths of a second to sleep.

If the sleeping process is awakened prematurely by a signal, tsleep() returns the number of ticks remaining to be slept.

See Also

`sleep()`; `F$Sleep` in the OS-9 Technical Manual.

ungetc()

Unget a Character

Synopsis

```
#include <stdio.h>

int ungetc(c, fp)
char c;                    /* character to unget */
FILE *fp;                  /* file pointer */
```

Function

`ungetc()` inserts the character c into the buffer associated with the file pointed to by fp. That character is returned by the next call to getc() on that file. `ungetc()` returns the character c and leaves the file fp unchanged.

One character of pushback is guaranteed provided something has been read from the file and the file is actually buffered.

If c is EOF (–1), ungetc() does nothing to the buffer and returns EOF.

Caveats

`fseek()` causes the pushed back character to be forgotten. A read from the file is required before ungetc() can push back a character. The file stdin, however, need not be read before a call to ungetc(). Only one character can be pushed back onto a file until a getc() is performed.

See Also

`getc()`, `fseek()`, and `setbuf()`

unlink()

Unlink (Delete) a File

Synopsis

```
int unlink(name)
char *name;               /* pointer to name of file to unlink */
```

Function

`unlink()` decrements the link count of the file whose pathname is name. When all the links to a file have been removed, the space occupied by the file on the disk is freed and the file ceases to exist.

If successful, unlink() returns 0. Otherwise, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

OS-9 does not yet support multiple links to a file. `unlink()` always causes the file to be removed from the disk. Attempting to delete a file that is open by the calling (or another) process results in a record-lock error.

See Also

`I$Delete` in the OS-9 Technical Manual.

unlinkx()

Unlink (Delete) a File

Synopsis

```
#include <modes.h>

int unlinkx(name, mode)
char *name;                 /* pointer to pathlist of file to unlink */
short mode;                 /* access permission */
```

Function

unlinkx() decrements the link count of the file whose pathname is name. If the mode passed indicates the execution directory, the path is assumed to be based in the current execution directory. The header file <modes.h> defines the legal mode values. When all the links to a file have been removed, the space occupied by the file on the disk is freed and the file ceases to exist.

If successful, unlinkx() returns 0. Otherwise, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

OS-9 does not yet support multiple links to a file. unlinkx() always causes the file to be removed from the disk. Attempting to delete a file that is open by the calling (or another) process results in a record-lock error.

See Also

I$Delete in the OS-9 Technical Manual.

wait()

Wait for Process Termination

Synopsis

```
wait(status)
unsigned *status;          /* pointer to exit status */

wait(0)
```

Function

`wait()` suspends the current process until a child process has terminated.

The call returns the process ID of the terminating process and places the exit status of that process in the unsigned int pointed to by status. If status is passed as zero, no child status is available to the caller.

The lower 16-bits of the status value contains the argument of the exit() or _exit() call as executed by the child process, or the signal number if it was interrupted with a signal. A normally terminating C program with no explicit call to exit() in main() returns a status of zero.

`wait()` returns –1 if there is no child process for which to wait, or 0 if a signal was received before a child process terminated.

Caveats

Note that the status codes used in OS-9 may not be compatible with other operating systems. A wait must be executed for each child process created.

See Also

`os9fork()`; `F$Wait` in the OS-9 Technical Manual.

write(), writeln()

Write Bytes to a Path

Synopsis

```
int write(path, buffer, count)
int path;                   /* path number */
char *buffer;               /* pointer to buffer to write */
unsigned count;             /* number of bytes to write */

int writeln(path, buffer, count)
int path;                   /* path number */
char *buffer;               /* pointer to buffer to write */
unsigned count;             /* number of bytes to write */
```

Function

`write()` writes bytes to a path. The path number path is an integer which is one of the standard path numbers 0, 1, or 2, or a path number returned from a successful call to open(), creat(), create(), or dup(). buffer is a pointer to space with at least count bytes of memory from which write() obtains the data to write to the path.

It is guaranteed that at most count bytes are written. If less bytes are written, an error has occurred.

`writeln()` causes **output-filtering** to take place such as outputting a linefeed after a carriage return, or handling the page pause facility. `writeln()` writes, at most, one line of data. The end of a line is indicated by a carriage return. `writeln()` is the preferred call for writing to the terminal.

`write()` essentially does a **raw** write, that is, the write is performed without translation of characters. The characters are passed to file (or device) as transmitted by the program.

`write()` and `writeln()` return the number of bytes actually written. If an error occurred, –1 is returned and the appropriate error code is placed in the global variable errno.

Caveats

Notice that writeln() stops when the carriage return is written, even if the byte count has not been exhausted. `write()` will do a *raw* write to the terminal; no linefeeds after return are transmitted.

See Also

`I$Write`, `I$WritLn` in the OS-9 Technical Manual.

OS-9/68000 Source Level Debugger

There are five chapters in this section of the manual. The first is an overview of Srcdbg. The next four introduce the various Srcdbg commands. A single example program is provided to introduce each of the commands. With this program, the user will have access to the actual source code while examining the Srcdbg command examples. The last chapter is included as a quick reference for Srcdbg commands and syntax.

The individual chapters include:

- Overview to Srcdbg: This chapter describes how to get in and out of Srcdbg. This includes instructions on setting the proper environment variables and the executive command line syntax. Srcdbg execution characteristics and caveats are described in detail. The scope and syntax of Srcdbg's command set are also described.
- Debugger Control Commands: These commands allow the user to control the execution of a program. They also control the communication between the debugger and the program.
- Data Manipulation Commands: These commands allow the user to access, examine and alter program data.
- System Commands: These commands allow the user access to the source files being debugged and the OS-9 shell.
- Assembly Level commands: These commands allow the user to control the execution of a program at the machine instruction level.

Chapter 6

Overview of SrcDbg

Overview of SrcDbg

In order for SrcDbg to work properly, the following three conditions must be met:

1. Current revisions of OS-9 and the C Compiler/Assembler/Linker must be used.
2. Some environment variables must be set.
3. A symbol file must be generated.

C Compiler Revision Requirements

In order to run SrcDbg, the "SrcDbg" file should be placed in your execution directory. The C Compiler, Assembler and Linker are necessary in order to generate the necessary files for debugging. These components must be the following revisions (or greater):

Component:	Revision:
C Compiler	Rev 3.0
R68	Rev 1.9 (edition 54)
R68020	Rev 2.9 (edition 77)
L68	Rev 1.9 (edition 53)

Setting the Environment

The PORT environment variable must be set in order to use SrcDbg. PORT specifies the terminal device name being used (e.g. /t1). SrcDbg uses PORT to determine the device on which it is running. PORT is automatically set by TSMON when logging onto a time-sharing system. If a time-sharing system is not being used, PORT must be set manually. For example, the following command sets PORT to /term:

```
setenv PORT /term
```

SrcDbg must also know in which directories to look for the compiled program, its symbol file and its source file(s). The search routine uses the SOURCE and PATH environment variables.

PATH and SOURCE are defined as follows:

Variable:	Description
PATH	Specifies the directories to be searched for the compiled program, its ".dbg" file, and its ".stb" file. For example: setenv PATH /h0/cmds:/h0/usr/walden/cmds:/dd/cmds
SOURCE	Specifies the directories to be searched for the source files of the program to be debugged. For example: setenv SOURCE .:/h0/usr/walden/work1:/h0/usr/walden/source

`SOURCE` and `PATH` do not have to be set in order to run SrcDbg. If they are set, they are used by SrcDbg when searching for the following files:

Compiled Programs
SrcDbg first looks in memory to see if the program has already been loaded into the module directory. If it is not found, the current execution directory is searched. If the program is not there either, SrcDbg searches the directories specified by the PATH environment variable. If it is still not found, an error will be returned.

Symbol Files
SrcDbg first searches the STB directory of the current execution directory. If the symbol file is not there or there is no STB directory, SrcDbg searches the STB directory located in each of the directories specified by the PATH environment variable. If the symbol file is not found in these directories, the current execution directory is searched. If the symbol file is still not found, each of the directories specified by the PATH is searched. If the symbol file is still not found, an error will be returned.

Source Files
SrcDbg first looks in the current data directory for the source file(s). If not found, SrcDbg searches the directories specified by the SOURCE environment variable. If the files are still not found, an error will be returned.

The ".dbg" and ".stb" Symbol Files

SrcDbg uses symbol files generated by the C Compiler and linker. The program must be compiled with the "–g" option to create these symbol files. For example:

```
cc program.c –g
```

This creates two symbol files, <program_name>.dbg and <program_name>.stb . The symbol files are created in the STB directory located in the user's current execution directory. If the STB directory does not exist, the symbol files are created in the user's execution directory. The ".dbg" and ".stb" symbol files are read automatically by SrcDbg when the debugging session begins.

Important: Using the "–g" option of the linker does not necessarily accomplish the same effect. "l68 –g" will create ".dbg" files only if the ".r" files were created with the "–g" option. In this case, they contain the information necessary to create the ".dbg" files. If they were not compiled with the "–g" option, only an ".stb" file is created.

Invoking SrcDbg

The syntax for the SrcDbg command line is:

```
srcdbg {<srcdbg_opts>} [<program>] {<program_opts>} [["]<redirections>["]]
```

`<srcdbg_opts>` are the SrcDbg command line options:

Option:	Description
–d	SrcDbg will not read the ".dbg" file
–m[=]<memory>	Extra memory will be allocated for <program>
–s	SrcDbg will not read the ".stb" file
–z[=]<pathlist>	SrcDbg will read commands from <pathlist>

`<program>` is the name of the C program to be debugged. <program_opts> are passed directly to <program> as command line arguments.

Important: `<options>` specified in the Fo[rk] command have the same effect as the <program_opts> specified on the SrcDbg command line. A full description of the Fo[rk] command is provided in the Debugger Control Commands chapter.

Shell filename wildcard processing can also be used. For example, in the following command the wildcards are expanded by the Shell and passed to SrcDbg, which in turn passes the filenames to "program":

```
srcdbg program *.c
```

The standard paths of both SrcDbg and the program may be redirected by specifying the redirected paths without enclosing quotes. For example:

```
srcdbg program >>>nil <data
```

The program's standard paths may be redirected by specifying the redirected paths within quotes. For example:

```
srcdbg program “>>>nil <data”
```

By redirecting the program's standard paths, I/O from SrcDbg remains unaffected. If SrcDbg's standard paths are redirected, all child Shells will inherit the redirected paths. This could cause problems when trying to execute the SrcDbg "shell" command.

Once SrcDbg is invoked it searches the current data directory for a file named SrcDbg.init and processes the SrcDbg commands in the file. A SrcDbg.init file is not required for use of SrcDbg, but it may be helpful if the user wants to do the same set of commands each time SrcDbg is invoked.

The SrcDbg.init file may include comments. Comments are indicated by the command * (an asterisk).

For example, a SrcDbg.init file may look like this:

```
option watch fpu           ;* turn off watch location and floating point register display
setenv MYENV “my program’s environment” ;* set an environment variable
setenv SOURCE C: TEST      ;* set source directories custom for current directory
setenv PATH /n0/richard/h0/cmds         ;* set PATH for future forks by this program
```

After the SrcDbg.init file has been processed, SrcDbg will process the specified command line arguments and fork the compiled program. If the compiled program is not found or could not be forked, the following error is returned:

```
Could not fork “<file>” – Error #000:216 (E$PNNF) File not found.
SrcDbg:
```

The "`SrcDbg:` " prompt indicates that SrcDbg is waiting for a command.

If SrcDbg is unable to fork the program, SrcDbg is not exited. Make sure the compiled program exists and is in your execution directory and try again.

After forking the program, SrcDbg searches for the ".dbg" and ".stb" files. If one or both of these symbol files are not found, the following error is returned:

```
Could not open symbol file “<symbol_file>”– Error
#000:216 file not found
SrcDbg:
```

If the symbol file is not found, SrcDbg is not exited. Make sure the missing symbol file(s) exists and is in your execution directory. SrcDbg can still be run without one or both of the symbol files. If there is no ".dbg" information, only assembly level commands will be of use. If there is no ".stb" information either, only assembly level commands will be of use, and SrcDbg will not have symbolic information, only addresses. If there is a ".dbg" file, but not an ".stb" file, symbols that are part of libraries which have not been compiled with the –g option will not be recognized.

When the symbol files are found, they are examined to verify that they match the code module being debugged. The CRC of the program module is stored in each symbol file.

If the CRC held in either symbol file does not match the CRC of the program module, the following prompt is returned:

```
Reading symbol file "<program_name>.dbg":
Symbol file "<symbol_file>" does not match object module "<module>"
Do you wish to use these symbols? (y or n):
```

This message indicates one of two problems. The symbol file was not generated at the same time as the program was compiled. This is usually due to an old version of the program existing in memory. If this is the case, answer "n" to the prompt, unlink the program (if no one is using it) and try again. If the program module was altered by the FIXMOD utility, the CRC of the module changes. The symbol file will still be current, but the CRC's will not match. If this was the case, answer "**y**" to continue.

Important: If a symbol file is used that does not actually match the program module, SrcDbg's behavior will be unpredictable.

After SrcDbg reads the program's symbol information, the context of the first executable instruction, usually located at _cstart, is displayed. In this example, the program where is searching for a file named dirt:

```
$ SrcDbg where dirt
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg:
```

Important: To reach the first executable source line, use the S[tep] command or B[reak] at that line or G[o] to that line. For example:

```
SrcDbg: step
File: "where.c"
19: main(argc,argv)
    ^
```

Important: Do not use the N[ext] command to get to the first executable source line. If the N[ext] command is used before the first executable source line is encountered, the program will run to completion. This is because the next executable source line in the current function (_cstart) does not exist.

When the program being debugged passes through a traphandler module or a subroutine module, SrcDbg will try to read the outside module's symbol file at that time. Then, the SrcDbg commands may be used in the outside module just as they are used in the main program. If symbol information is not found, SrcDbg's assembly level commands may still be used in the outside module.

SrcDbg provides an execution pointer indicating the current place of execution of the program. The circumflex pointing to main in the example above indicates the beginning of execution for the program "where".

The execution pointer will stop at each of the following:

1. comma operator
2. logical AND (&&) or logical OR (||)
3. ternary operator (?:)
4. expression(s) of the following:
    a. while loop
    b. do loop
    c. for loop
    d. if statement
    e. switch statement
    f. return statement
5. initializer
6. expression statement

Consequently, when stepping through a program, SrcDbg may redisplay the same line of source code and merely move the execution pointer to the next execution position.

Use of MACROS in source code may cause the SrcDbg execution pointer to become temporarily misaligned. The compiler system expands the MACRO and gives the expanded code information to SrcDbg. SrcDbg will display the correct, but unexpanded source line. The execution pointer will try to indicate each of the expanded executable statements. This causes the execution pointer to become temporarily misaligned.

SrcDbg Help

Once the SrcDbg session is begun, "on-line" help is available using the HELP command. Type "**help**" or "**?**".

SrcDbg will look for the HELP file in /dd/sys. If it is not found, the following error is displayed:

```
Could not open file "/dd/sys/SrcDbg.hlp" - Error 000:216
SrcDbg:
```

Exiting SrcDbg

SrcDbg may be exited at any point by entering the Q[uit] command or the **[Esc]** key at a SrcDbg prompt. SrcDbg will close all necessary files and return control to the Shell. The symbol files are NOT deleted, allowing you to debug the same module at a later date.

SrcDbg Command Syntax

Each SrcDbg command can be invoked using a long or a short form. Throughout this manual the long form is shown within brackets appended to the short form (i.e. L[ist]). For example, the following two L[ist] commands accomplish the same result:

```
SrcDbg: l program.c
SrcDbg: list program.c
```

Most commands accept command line parameters. Because of the nature of a source level debugger, the parameters often take the same form. The three most important parameters are <scope_expr>, <line_num>, <C_expr> and <location_expr>. These parameters will be described in more detail later in this chapter.

SrcDbg Scope

Each program object has a scope. Scope is defined as the block in which the object is known and the blocks which can access the object. In general, this means the block containing the object and all blocks contained within that block.

As SrcDbg reads the program's symbol files, SrcDbg creates a program block "tree". This tree helps define the scope of SrcDbg commands at any specific time. To illustrate this, a debug process and its associated block tree are listed in the next two pages. These files are included on the distribution disk and may be compiled to enable the user to try the examples.

In this illustration, there are several variables named "i". Each of these has a different scope.

Important: When more than one program object have the same name and overlap in scope, the innermost object is assumed.

Debug Process Source Code

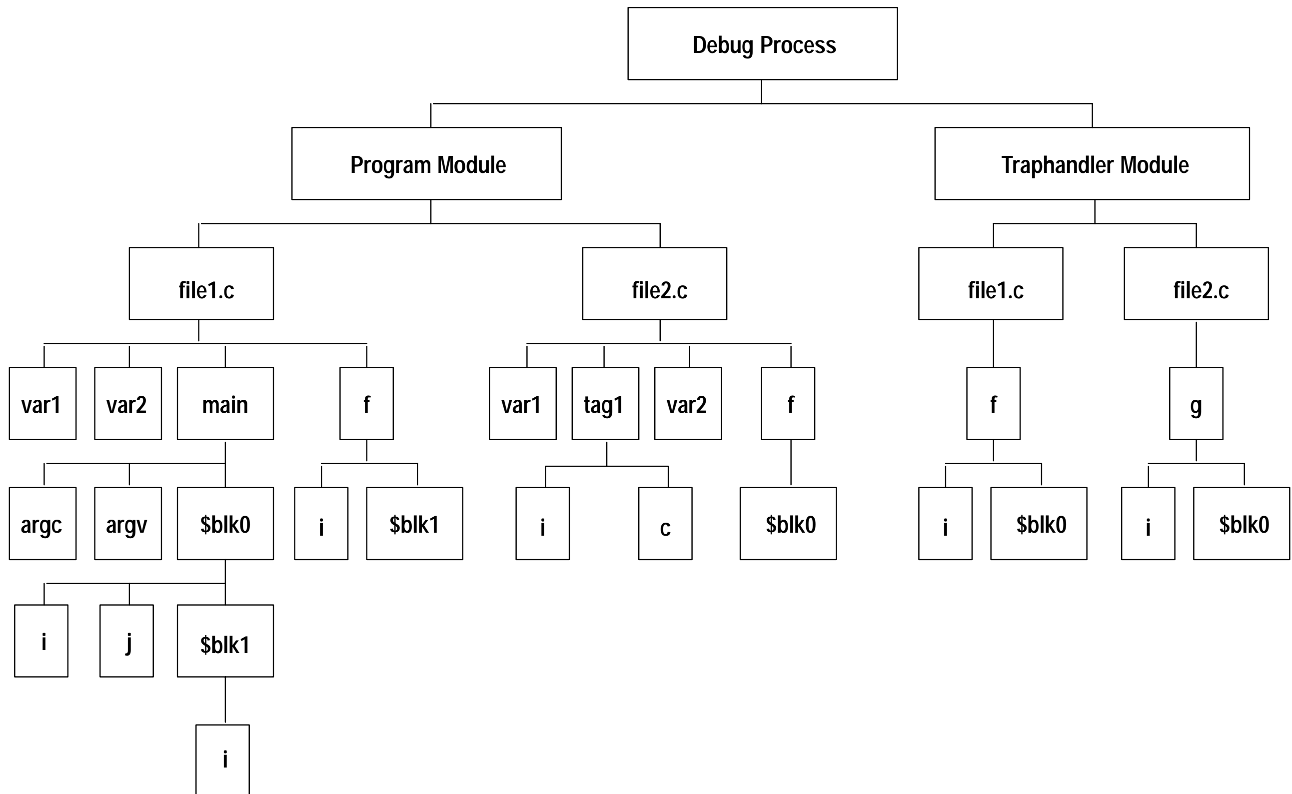
Example program:

```
program\file1.c            program\file2.c

1  int var1;  1  static float var1;
2  int var2;  2  struct tag1 {
3             3      int i;
4  main(argc,argv)         4      char c;
5  int argc;  5  }var2;
6  char *argv[];           6
7  {/* block 0 */          7  static f()
8    extern void f();      8  {
9    int i;                               9     int i;
10   static int j;                        10    i = 0;
11                                        11 }
12   g(i = argc);                         12
13   if(i > 1) { /* block 1 */
14       int i;
15       i = i;
16       printf("i = %d\n",i);
17       f(i);
18   }
19   return i;
20 }
21
22 void f(i)
23 int i;
24 {
25     print("%d\n",i);
26 }
```

Example traphandler:

```
traphandler\file1.c        traphandler\file2.c

1  int f(i)   1  int g(i)
2  register int i;         2  register int i;
3  {          3  {
4      return i;           4      return i;
5  }          5  }
```

Figure 9.1
Process Block Tree



Scope Expressions

The scope of a command may be specified by using a scope expression. A scope expression has the following syntax:

```
<module_name>[\<source file>]{\<identifier>}
or
<source_file>{\<identifier>}
or
<identifier>{\<identifier>}
```

A special form of the scope expression can be used by the I[nfo] command:

```
<module_name>[\<source file>]{\<identifier>}[\<struct_union_enum_specifier>]
or
<source_file>{\<identifier>}[\<struct_union_enum_specifier>]
or
<identifier>{\<identifier>}[\<struct_union_enum_specifier>]
```

The <struct_union_enum_specifier> takes the form of:

```
enum <tag_identifier>
or
struct <tag_identifier>
or
union <tag_identifier>
or
<struct_identifier>[.<field_identifier>]
or
<union_identifier>[.<field_identifier>]
```

The following commands allow the user to specify the scope in which to execute the command: L[ist], B[reak], I[nfo], P[rint], A[ssign], W[atch], C[h]c, Con[text], G[o], C[hange], Di[sasm], Dil[ist], Mf[ill], Ms[earch], Sy[mbol] and D[ump]. If a scope is not specified, by default, the current scope is used. The current scope is the current block and all blocks containing it. The current scope may be changed with the C[h]c command.

Using the example in Figure 9.1 and the I[info] command, the following example scope expressions could be executed:

```
SrcDbg: info program\file2.c\var1
static float var1;

SrcDbg: info program\file1.c\f
void f(i)
int i;

SrcDbg: info program\file2.c\struct tag1
struct tag1 {
int i;
char c;
};

SrcDbg: info traphandler\file1.c\tentry\i
register int i;
```

SrcDbg allows the user to specify objects within program blocks by using the notation “$blk<num>” for the block number within a program. The left bracket following a compound statement begins each new block. The associated right bracket ends the block. Blocks are successively numbered from the beginning of each function starting with block 0 ($blk0). Consequently, to refer to the variable “i” on line 17 the following full scope expression would be used:

```
SrcDbg: info program\file1.c\main\$blk0\$blk1\i
int i;
```

SrcDbg does not always need full scope expressions. For example, if you were at line 17 in “program\file1.c”, the command “info i” would reference the “i” declared within block one ($blk1). To reference the variable “i” declared in block zero ($blk0), you could use any of the following commands:

```
info program\file1.c\main\$blk0\i
info file1.c\main\$blk0\i
info main\$blk0\i
info $blk0\i
```

When referencing static variables, types, structs, unions and enums that are not in the current file, the appropriate file must be referenced (i.e. <file>\... or <module>\<file>\...). Similarly, when referencing global symbols that are not part of the current module, the appropriate module must be referenced (i.e. <module>\<identifier>) .

Line Number Expressions

Many commands use the <line_num> parameter. This specifies a line in a specific program by its line number. The syntax for the line number expression is:

```
<number>
```
or
```
<file>\<number>
```
or
```
<module_name>\<file>\<number>
```

If the “<file>\” format is not used, the current file is assumed to hold this line <number>.

For example, using the previous program and the L[ist] command, the following line number commands could be used:

List a file from line 7 to eof:

```
SrcDbg: list program\file2.c\7
   7: static f()
   8: {
   9:    int i;
  10:    i = 0;
  11: }
  12:
[end of file]
```

List a range:

```
SrcDbg: list program\file1.c\7 12
   7: {  /* block 0 */
   8:     extern void f();
   9:     int i;
  10:     static int j;
  11:
  12:     g(i = argc);
```

List a function:

```
SrcDbg: list traphandler\g
   1:  int g(i)
   2:  register int i;
   3:  {
   4:      return i;
   5:  }
[end of file]
```

C Expressions

All C language expression operators are implemented with the following exceptions:

Operator:	Description
?	ternary operator
++	increment operator
=	assignment operator
––	decrement operator
,	comma operator

Strings are not supported in C expressions. Character constants, however, are supported. All supported C expressions are evaluated with standard precedence.

Function calls may also be used in C expressions. For example:

```
SrcDbg: print f(i)
```

Limited "casting" is supported by SrcDbg. The following types may be cast:

```
<type>
<pointer to struct>
<pointer to union>
<enum>
```

The supported "cast" syntax for these are, respectively:

```
             <type>  :=  <type_name> {*}
<pointer to struct>  :=  struct <tag_name> * {*}
 <pointer to union>  :=  union <tag_name> * {*}
             <enum>  :=  enum <tag_name> {*}
```

Register names may also be used in C expressions. The following registers are supported for the MC68xxx:

Register:	Description
.d0 - .d7	data registers
.a0 - .a7	address registers (.sp may be used for .a7)
.sr	lower order byte of the status register (.cc may be used for .sr)
.pc	program counter

These registers are supported for the MC68881 floating point coprocessor:

Register:	Description
.fp0 – .fp7	floating point registers
.fpcr	control register
.fpsr	status register
.fpiar	instruction address register

SrcDbg also supports the use of debugger convenience registers .r0 - .r7 . These registers are available so that the user may store expression results for later access. They are commonly used with the D[ump] command.

Location Expressions

A <location_expr> is a combination of a <C_expr> and a <line_expr>. If <location_expr> is specified, the program will run until <location_expr> is reached. A number in <location_expr> is interpreted as a line number, unless preceded by an @ (“at” sign). Then, the number will be interpreted as an absolute location.

If a function name is alone in a location expression, and there is source level information about the function, the location specified will be the first source line of the function. If a location is required at the first instruction of the function, an absolute location must be provided. The absolute location may be determined with the print command. (i.e. print <function_name>).

If a function name is alone in a location expression and there is no source level information for the function, the location will be the first instruction of the function.

Location expressions are used by the B[reak], G[o] and Dil[ist] commands.

For example, the following command would be used to break at the absolute location 0xce042:

```
SrcDbg: break @0xce042
```

This command would be used to execute the program until line 112 in the current file:

```
SrcDbg: go 112
```

This command would be used to Dil[ist] starting at the location printer+0x16:

```
SrcDbg: dilist printer + 0x16
```

Command Line Notes

The S[tep], N[ext], L[ist], T[race], Dil[ist], Gostop and D[ump] commands each may be executed repeatedly by subsequent <returns>. This is indicated by the command name included as part of each subsequent prompt. For example:

```
SrcDbg(STEP):
```

When the Fo[rk], D[ump], Fi[nd], Re[ad], C[hange], Di[sasm], Dil[ist], Sy[mbol] or P[rint] command is executed with no arguments, the last command line respectively is repeated. For example, if the following Fo[rk] command is executed:

```
SrcDbg: fork where where.c /h0/usr/walden
```

Each subsequent Fo[rk] command that does not specify any command arguments would then execute the above command line.

All commands may be sequentially executed on the same command line by using semicolons to separate the individual commands from each other. For example:

```
SrcDbg: break read_dir_stuff; go; step; step; step
```

All commands may be aborted using the keyboard interrupt signal (control-C) or the keyboard quit signal (control-E).

If a command begins with a * (star), the line will be interpreted as a comment. This may be useful to add comments to SrcDbg.init files, L[o]g files, or any other command files.

Example Tutorial Program

The following program will be used throughout this manual in examples to show the use of commands. This program may be copied and compiled on your own system. Its function is to search specified directories for specified files.

```
where.c
  1   #include <stdio.h>
  2   #include <modes.h>
  3   #include <errno.h>
  4   #define BUFSIZE 512
  5
  6   typedef enum {
  7       FALSE,
  8       TRUE
  9   } boolean;
 10
 11   void    main(argc,argv), check_args(argc),
 12           fork_dir(argv,pathlist,pid,pipe,stout),
 13           read_dir_stuff(filename,pid,pipe,stout),
```

```
14          sighand(), kill_dir(pid);
15  boolean print_args(argv);
16  char    inbuf[BUFSIZE];
17  boolean sigflag = FALSE;
18
19  void main(argc,argv)
20  register int        argc;
21  char            *argv[];
22  {
23      register char   *filename = argv[1];
24      auto int        pid,pipe,stout;
25
26      check_args(argc);
27      fork_dir(argv,argv[2],&pid,&pipe,&stout);
28      read_dir_stuff(filename,pid,pipe,stout);
29  }
30
31
32  void check_args(argc)
33  register int argc;
34  {
35      if(argc < 2) exit(_errmsg(0,“where <filename> [<<pathlist>]\n”));
36  }
37
38
39  void fork_dir(argv,pathlist,pid,pipe,stout)
40  register char       **argv,
41                      *pathlist;
42  register int        *pid,
43                      *pipe,
44                      *stout;
45  {
46      register int    stin;
47      extern int      os9fork(), os9exec();
48      extern char     **environ;
49
50      /* initialize argv list for dir */
51      argv[0] = “dir”;
52      argv[1] = “-rasu”;
53      argv[2] = pathlist;
54      argv[3] = NULL;
55
56      /* dup standard out “save it” and close standard out */
57      *stout = dup(1);
58      close(1);
59
60      /* open /pipe - now standard out - dir will write to pipe */
61      if((*pipe = open(“/pipe”,(S_IWRITE | S_IREAD))) == -1)
62          exit(_errmsg(errno,“can’t open pipe - ”));
63
64      /* fork dir with extra stack */
65      if((*pid = os9exec(os9fork,*argv,argv,environ,10240,0,0)) == -1)
66          exit(_errmsg(errno,“can’t fork dir - ”));
67  }
```

```
 68
 69
 70   void read_dir_stuff(filename,pid,pipe,stout)
 71   register char       *filename;
 72   register int        pid,pipe,stout;
 73   {
 74       register char   *buf = inbuf,*str,*p;
 75
 76       /* install intercept */
 77       intercept(sighand);
 78
 79       /* close stdin, dup open /pipe, close stdout, dup old stdout */
 80       close(0);
 81       dup(pipe);
 82       close(1);
 83       dup(stout);
 84
 85       /* we will read the pipe (dir’s output) */
 86       while(gets(buf)) {
 87           if(str = (char *)rindex(buf,’/’)) {
 88               *str = ’\0’;
 89               p = str + 1;
 90           } else p = buf;
 91           if(!strcmp(p,filename)) puts(buf);
 92           if(sigflag) break;
 93       }
 94       if(sigflag) kill_dir(pid);
 95   }
 96
 97   void sighand()
 98   {
 99       sigflag = TRUE;
100   }
101
102   void kill_dir(pid)
103   register int    pid;
104   {
105       if(setuid(0) == –1) exit(_errmsg(errno,“could not do setuid –
”));
106       if(kill(pid,2) == –1) exit(_errmsg(errno,“could not signal – ”));
107   }
108
109   #ifdef DEBUG
110   boolean print_args(argv)
111   register char        **argv;
112   {
113       register int    i = 0;
114
115       while(*argv) fprintf(stderr,“%d: %s\n”,++i,*argv++);
116       return i >= 2;
117   }
118   #endif
```

Notes

Chapter 7

Debugger Control Commands

The commands described in this chapter control the execution of SrcDbg. They provide the following functions:

Command:	Description:
Fo[rk]	starts a program from within SrcDbg
G[o]	executes the program until a breakpoint occurs or the program stops
S[tep]	executes one statement at a time
N[ext] S[tep]	executes one statement at a time. N[ext] does not, however, step through functions. Instead it executes at full speed until the function returns.
R[eturn]	executes the program until returning to the calling function
B[reak]	sets breakpoints in the program
W[atch]	specifies program objects to be monitored during execution of the program
K[ill]	removes breakpoints and watch expressions in the program
L[o]g	writes SrcDbg commands to the specified pathlist
O[ption]	allows the user to change certain SrcDbg options
Re[ad]	reads SrcDbg commands from the specified pathlist

Fork

Start Program

Syntax

```
fo[rk] <program> [<program_opts>] [["]<redirection>["]]
```

Usage

`Fo[rk]` begins execution of the specified program. Once forked, the debug process' program counter will be pointing to the primary module's execution entry point (usually at _cstart). Any <program_opts> appearing on the Fo[rk] command line will be passed to the program as parameters. If there are no arguments specified, Fo[rk] re-starts the program with the same arguments used by either the last Fo[rk] command or the SrcDbg command line.

`<redirection>` modifiers may also be specified on the Fo[rk] command line. This allows lengthy I/O to go to or originate from a file or device other than your terminal. If the redirected paths are given within quotes, only the standard paths of the program being debugged are affected.
For example:

```
SrcDbg: fork where where.c /h0 ">>>foundit"
```

If the redirected paths are specified without quotes, the standard paths of SrcDbg and the program being debugged are redirected. For example:

```
SrcDbg: fork where where.c /h0 <>>>/nil
```

This type of redirection affects all consequent Shell commands executed from SrcDbg. In the above example, it is consequently impossible to fork a Shell from SrcDbg.

The Fo[rk] command may be given at anytime. This effectively ends the current debugging session and begins a new session.

Examples

Each of the following three examples accomplish the same result: forking the program where; passing two parameters to where ("where.c" and "/h0/usr/kathie"); executing the G[o] command to run where
to completion.

In the first example, the program where is forked from the SrcDbg command line with no parameters. The G[o] command is given to run where to completion.

```
$ SrcDbg where where.c /h0/usr/kathie
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: go
/h0/usr/kathie/SEA
/h0/usr/kathie/SOURCE
"where" exited normally
```

"where" forked with 2 parameters

execute where (g command)

`where` is now re-forked with the Fo[rk] command. The two parameters ("where.c " and "/h0/usr/kathie") are passed to where as command line arguments.

```
SrcDbg: fork where where.c /h0/usr/kathie
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg:go
/h0/usr/kathie/SEA
/h0/usr/kathie/SOURCE
"where" exited normally
```

re–fork "where"

"where" output

`where` is now re-forked with the Fo[rk] command. Because no parameters are specified, the most recent Fo[rk] command line is used by default ("fork where where.c /h0/usr/kathie").

```
SrcDbg: fo
Forked: "where"
Context: where\_cstart
SrcDbg:go
/h0/usr/kathie/SEA
/h0/usr/kathie/SOURCE
"where" exited normally
```

re–fork "where" with last arguments used

"where" output

Go

Program Execution

Syntax

```
g[o] [<location_expr>] [:dis[play]]
```

Usage

`G[o]` executes the program starting at the current location. `G[o]` executes the program until:

- a breakpoint is encountered
- an exception occurs
- a signal occurs, e.g. keyboard interrupt ([Ctrl-C]), keyboard abort ([Ctrl-E]), etc.
- the end of the program is reached

If ":dis[play]" is specified, SrcDbg displays each source line to be executed.

Important: If watch expressions have been set, SrcDbg will evaluate each watch expression and display its value if it changes after executing each C statement. For example:

```
SrcDbg: go
w1: argc
2
```

Important: Using Watch expressions or Break when expressions while executing a program with the G[o] command will slow execution speed.

Examples

In the following example, where is forked with two parameters. These are passed to where as command line arguments. The G[o] command is then given to run where to completion.

```
$ SrcDbg where where.c /h0/usr/kathie                    fork where
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: go                                               execute where (g command)
/h0/usr/kathie/SEA                                       “where” output
/h0/usr/kathie/SOURCE
“where” exited normally
```

`where` is re-forked and a breakpoint is set at line 37 using the B[reak] command. The G[o] command is then given with the “:dis” option. Once the breakpoint is reached, argv[0] is specified as a watch expression. The G[o] command is executed again and where runs to completion displaying the changing values of argv[0]:

```
SrcDbg: fork where /h0/usr/kathie                        re–fork “where”
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: go main                                          execute to main
File: “where.c”
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg: break 39                                         set breakpoint at line 39
b1: where\where.c\39 :count 1
SrcDbg: go :dis                                          execute in display mode until breakpoint

  23:   register char   *filename = argv[1];
                                           ^
  26:   check_args(argc);
        ^
  32: void check_args(argc)
           ^
  35:   if(argc < 2) exit(_errmsg(0,“where <filename> [<pathlist>]\n”));
           ^
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
        ^
At breakpoint #1                                         breakpoint at line 39 encountered
File: “where.c”
```

```
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
SrcDbg: watch argv[0]
w1: argv[0]
SrcDbg: go
Watch Expression #1: argv[0]
0x30940 = "where"
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
dn: FFFFFFA8 00000000  00000002 00000003   0003085C 00030858  00001980 00000000
an: 00030948 00000000  00030948 00000000   00030860 00030844  00037000 00030820
pc:    E6A9E  cc:  08 (-N---)
fork_dir+0x1e       >6100FE32         bsr.w _stkchec
Watch Expression #1: argv[0]
0xe6c7f = "dir"
File: "where.c"
Context: where\fork_dir\$blk0
  52:   argv[1] = "-rasu";
        ^
"where" exited normally
```

watch expression's initial value
watch location display

watch expression changes and shows instruction immediately following the one that changed it

(there is no assembly level display since instruction is exactly on a source line)

Step

Single Line Execution

Syntax

```
s[tep] [<number>]
```

Usage

`S[tep]` executes the specified number of executable statements of the program. If no number is specified, S[tep] executes a single statement. SrcDbg executes a statement, displays the values of any watch expressions that have changed and displays the next executable line. SrcDbg then displays the following prompt:

```
SrcDbg(STEP):
```

To continue "stepping" through a program, <return> may be used as well as re-entering the S[tep] command. If a number was previously specified, it remains in effect until a S[tep] command with a different number (or no number) is executed.

If a function is encountered while stepping through a program, the function is also stepped through. To avoid stepping through functions, use the N[ext] command.

Important: The exception to this is a function that was not compiled with the program. For example, any of the OS-9 standard C library functions. These types of functions will be executed and stepped over with a single S[tep] command.

If a breakpoint is encountered while stepping through a program, execution stops in the same manner as when using the G[o] command.

Examples

In the following example, where is forked from the SrcDbg command line with two parameters. These are passed to where as command line arguments. The S[tep] command is given to execute the first statement. The S[tep] command is repeated in order to show its use.

```
$ srcdbg where where.c /h0/usr/kathie                    "where" forked
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: step
File: "where.c"
  19: void main(argc,argv)
           ^
SrcDbg: step
  23:   register char   *filename = argv[1];
                                          ^
SrcDbg(STEP):s                  function "check_args()" encountered
  26:   check_args(argc);
        ^
SrcDbg(STEP): step 3                                     step 3 times
  32: void check_args(argc)
           ^
  35:   if(argc < 2) exit(_errmsg(0,"where <filename> [<path>]\n"));
           ^
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
        ^
SrcDbg(STEP): <return>                             step 3 times again
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
  51:   argv[0] = "dir";
        ^
  52:   argv[1] = "-rasu";
        ^
SrcDbg(STEP):
```

Name

Single Line Execution/Execute Function and Return

Syntax

```
n[ext] [<number>]
```

Usage

`N[ext]` executes the specified number of executable statements of the program. If a number is not specified, SrcDbg executes one statement. SrcDbg executes a statement, displays the value of any watch expressions that have changed and displays the next executable line. SrcDbg then displays the following prompt:

```
SrcDbg(NEXT):
```

To continue “nexting” through the program, hit **[CR]** or re-enter the N[ext] command.

If a <number> is specified, it remains in effect until a N[ext] command with a different number (or no number) is executed.

The N[ext] command has the same function as the S[tep] command with one exception: if a function is encountered while “nexting” through a program, the function is executed without stepping through it. To step through functions, use the S[tep] command.

Important: The N[ext] command will execute a function and all functions called by that function and then display the next executable line of the program. This includes functions recursively calling themselves.

If a breakpoint is encountered while “nexting” through a program, execution stops in the same manner as when using the G[o] command. Execution will stop even if the breakpoint is within a function.

Examples

In the following example, where is forked with two parameters. These are passed to where as command line arguments. The N[ext] command is given to execute the first statement. This command is repeated to show its use. Notice the difference in function handling between the S[tep] command and the example below.

```
$ srcdbg where where.c /h0/usr/kathie                           “where” forked
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: go main                                                 execute to main
File: “where.c”
  19: void main(argc,argv)
           ^
SrcDbg: next
  23:   register char   *filename = argv[1];
                                          ^
SrcDbg(NEXT):n                        function “check_args()”  encountered
  26:   check_args(argc);
        ^
SrcDbg(NEXT): next 3                   next 3 functions without stepping
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
        ^
  28    read_dir_stuff(filename,pid,pipe,stout);
        ^
/h0/usr/kathie/SEA                                              where output
/h0/usr/kathie/SOURCE
“where” exited normally
```

Return

Execute Until Function Returns

Syntax

```
r[eturn] <number>
```

Usage

R[eturn] executes the program until the current function returns to the calling function or a breakpoint is encountered. If <number> is specified, R[eturn] executes until it returns to the specified <number> of callers above the current stack frame.

To find out the number of callers in the current stack frame, the Fr[ame] command is used. If more callers are specified than exist, an error is returned and the R[eturn] command is not executed:

```
Error:  Bad frame number
```

Important: If the frame to “return” to has no source, SrcDbg will run until the first source line is encountered. This is usually in a higher frame.

Examples

In the following example, where is forked with two parameters. These are passed to where as command line arguments. A breakpoint is set at the function “fork_dir” using the B[reak] command. The G[o] command runs where until “fork_dir” is reached. The S[tep] command is used to step through a few lines of fork_dir(). The R[eturn] command is then used to return from fork_dir(). The R[eturn] command is used again to return from main().

Session	Comment
`$ SrcDbg where where.c /h0/usr/kathie`	“where” forked
`Reading symbol file “where.dbg”.`	
`where.c`	
`Reading symbol file “where.stb”.`	
`Context: where_cstart`	
`SrcDbg: break fork_dir`	set breakpoint at fork_dir()
`SrcDbg: break fork_dir`	
`b1: where\fork_dir+0x18 :count 1`	
`SrcDbg:go`	execute until fork_dir() encountered
`At breakpoint #1`	breakpoint at fork_dir() encountered
`File: “where.c”`	
`Context: where.c\fork_dir`	
`39: void fork_dir(argv,argv[2],&pid,&pipe,&stout);`	
`^`	
`SrcDbg: step`	step into fork_dir()
`51: argv[0] = “dir”;`	
`^`	
`SrcDbg(STEP): step`	
`52: argv[1] = “-rasu”;`	
`^`	
`SrcDbg(STEP): <return>`	execute until return from
`30 read_dir_stuff(filename,pid,pipe,stout);`	fork_dir()
`^`	
`SrcDbg:r`	execute until return from main()
`/h0/usr/kathie/SEA`	
`/h0/usr/kathie/SOURCE`	(main returns to _cstart, which has no source,
`“where” exited normally`	so the program runs to completion.)

Break

Set Breakpoint

Syntax

```
b[reak] [<location_expr>] [:wh[en] <C_expr>] [:co[unt] <num>]
```

Usage

Breakpoints are generally used to stop execution in a program to allow single-line-stepping through specific areas.

`B[reak]` sets a breakpoint at a specified line number, result of a <C_expr> or upon a when <C_expr> becoming true.

The following chart outlines the possible configurations of the B[reak] command. x indicates that the corresponding B[reak] command line option is being used.

Location expr:	When expr:	Count (n)"	Result:
o	o	o	Displays established breakpoints
o	o	x	Error
o	x	o	Break when when_expr is true
x	o	o	Break at location_expr
o	x	x	Break when when_expr is true the nth time
x	o	x	Break at location_expr then nth time
x	x	o	Break at location_expr when when_expr is true
x	x	x	Break at location_expr when when_expr is true the nth time

When a breakpoint is encountered, SrcDbg prints the breakpoint number, the current program scope and the next line (containing the breakpoint).

Up to 16 breakpoints may be set at one time. However, SrcDbg sets and removes breakpoints in the execution of R[eturn], N[ext], Gostop, G[o] and P[rint] (when printing functions). If sixteen breakpoints are set by the user, SrcDbg will be unable to execute these commands until a breakpoint is removed. To remove a breakpoint, the K[ill] command is used.

If no arguments are specified with the B[reak] command, all current breakpoints in the program are displayed.

If <num> is not specified with the count option, count defaults to 1.

Each time a breakpoint is specified, SrcDbg checks to see if a breakpoint has already been set at that location. If a B[reak] command specifies a breakpoint at a non-executable line number, SrcDbg places the breakpoint at the closest executable line following the specified line number. If a line number is specified greater than the last executable line, an error is returned.

The following command line causes the debugger to stop at 'func' the third time 'arg_to_func' is equal to 10:

```
break func :when arg_to_func == 10 :count 3
```

The debugger will stop at 'func' and evaluate the expression. If the expression is true (non 0) and the count is 1, SrcDbg will stop. If the expression is true and the count is not 1, the count is decremented and SrcDbg will continue. If the expression is not true, SrcDbg will continue without decrementing count.

When count reaches 1, SrcDbg will stop each time the breakpoint is reached and will no longer decrement count.

A break point comprised of a when expression without a break expression will cause SrcDbg to execute slowly. When SrcDbg encounters the breakpoint, the when expression is true. If the G[o] command is executed next, SrcDbg will generally stop on the next instruction since the when expression will still be evaluated as true.

Examples

The following example shows how B[reak] may be used to set several types of breakpoints in the program where:

```
SrcDbg where where.c /h0/usr/kathie                          "where" forked
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: b fork_dir                                           set breakpoint at fork_dir
b1: where\fork_dir+0x10 :count 1
SrcDbg: g                                                    execute "where"
At breakpoint #1                                             stop at breakpoint
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
             ^
SrcDbg: chc read_dir_stuff\$blk0                             change context
SrcDbg: b 91 :when *p == 's'                                 set breakpoint at line 91
b2: where\where.c\91 :when *p == 's' :count 1                when p points to 's'
```

```
SrcDbg: g                                                   execute "where"
At breakpoint #2                                            stop at breakpoint 2
File: "where.c"
Context: where\read_dir_stuff\$blk0\$blk1
  91:       if(!strcmp(p,filename)) puts(buf);
               ^
SrcDbg: print p                                             display p
0x48036 = "scope_expr.bnf"
SrcDbg: kill b2                                             remove breakpoint 2
SrcDbg: break 87 :count 100                  set breakpoint at line 87 the 100th time
b2: where\where.c\87 :count 100
SrcDbg: g                                                   execute "where"
/h0/usr/kathie/SEA
/h0/usr/kathie/SOURCE
At breakpoint #2                                            stop at breakpoint 2
File: "where.c"
Context: where\read_dir_stuff\$blk0\$blk1
  87:       if(str = (char *)rindex(buf,'/')) {
               ^
SrcDbg: kill b2                                             remove breakpoint 2
SrcDbg: break :when .cc & 1                                 break when carry bit is set
b2:  :when .cc & 1 :count 1
SrcDbg: break                                               display breakpoints
#  Breakpoint:                    Condition:                    Count:
---------------------------------------------------------------------
 1 where\fork_dir+0x10                                          1
 2                                .cc & 1                       1
SrcDbg: go                                                  execute "where"
At breakpoint #2                                            stop at breakpoint 2
File: "where.c"
Context: where\rindex                                       note extended display
  28:   read_dir_stuff(filename,pid,pipe,stout);
        ^
dn: 0004801E 0000002F  0004821E 000492C0   0000002F 00000001  00000003 00000000
an: 002225B0 003EE29A  0004801F 0004801E   00048036 00049178  00050000 00049138
pc:   22297E  cc:  09 (-N--C)
<68881 in Null state>
rindex+0xc          >6602            bne.b rindex+0x10->
SrcDbg:
```

Watch

Set Watch Expression

Syntax

```
w[atch] [<C_expr>]
```

Usage

`W[atch]` monitors the value of specified <C_expr> as the program is executed. Each watch expression is evaluated each time a machine instruction is executed. The watch expression will be displayed with its initial value when it is first set. SrcDbg will then display each watch expression only when its value changes:

```
SrcDbg: go
w1: argv[0]
0x23e577 = "dir"
```

Two modes of W[atch] output are available with the O[ption] command. By default, SrcDbg will show the location of each watch expression change. The location displayed when a watch expression changes equates to the machine instruction after the change.

The o[ption] watch may be used to suppress the location display when a watch expression changes.

SrcDbg evaluates each expression being monitored after each machine instruction is executed. Consequently, using watch expressions while executing a program with the G[o] command will slow execution speed.

Up to 16 expressions may be monitored at any given time.

Important: SrcDbg is unable to monitor arrays, structs or unions. SrcDbg can only monitor the individual elements.

Examples

In the following example, where is forked with two parameters. These are passed to where as command line arguments. A breakpoints is set at fork_dir. where is run to fork_dir using the G[o] command. A watch expression is set and a W[atch] command with no parameters is then executed to verify the watch expression. The G[o] command runs where to completion. Note each time the watch expression changes:

```
$ SrcDbg where where.c /h0/usr/kathie                         "where" forked
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: step
File: "where.c"
  19: void main(argc,argv)
           ^
SrcDbg(STEP): break fork_dir                                  set breakpoint
b1: where\fork_dir+0x18 :count 1
SrcDbg: go
At breakpoint #1                                              stop at breakpoint
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
SrcDbg: watch fork_dir\argv[0]                                set watch expression

w1: fork_dir\argv[0]
SrcDbg: watch                                                 display watch expression

#  Watch Expression:
--------------------
 1 fork_dir\argv[0]
SrcDbg: go
Watch Expression #1: fork_dir\argv[0]                         watch expression's initial value
0x2f94c = "where"                                             watch location display
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
dn: FFFFFFA8 0002F890  00000003 00000003   0002F860 0002F85C  00001990 00000000
an: 0002F954 00000000  0002F954 0002F890   0002F864 0002F848  00036000 0002F824
pc:    E6A9E  cc:  08 (-N---)
fork_dir+0x1e       >6100FE32        bsr.w _stkchec
Watch Expression #1: fork_dir\argv[0]                         watch expression changes
0xe6c7f = "dir"
File: "where.c"
Context: where\fork_dir\$blk0                                 (there is no assembly level  display since
  52:   argv[1] = "-rasu";                                    instruction is exactly on a source line)
        ^
/h0/usr/kathie/SEA                                            where output
/h0/usr/kathie/SOURCE
"where" exited normally
```

This example is identical to the previous example except that the O[ption] watch command is used:

```
$ srcdbg where where.c /h0/usr/kathie                     fork “where”
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: o watch                         turn off watch location display
SrcDbg: step
File: “where.c”
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg(STEP): break fork_dir                            set breakpoint
b1: where\fork_dir+0x18 :count 1
SrcDbg: go
At breakpoint #1                                    stop at breakpoint
File: “where.c”
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
SrcDbg: watch fork_dir\argv[0]                    set watch expression
w1: fork_dir\argv[0]
SrcDbg: watch                                 display watch expression
#  Watch Expression:
--------------------
 1 fork_dir\argv[0]
SrcDbg: go
Watch Expression #1: fork_dir\argv[0]        watch expression changes
0x2994c = “where”
Watch Expression #1: fork_dir\argv[0]        watch expression changes
0x13bc7f = “dir”
/h0/usr/kathie/SEA                                      “where” output
/h0/usr/kathie/SOURCE
“where” exited normally
SrcDbg:
```

Kill

Remove Breakpoint/Watch Expression

Syntax

```
k[ill] [<breakpoint>] {[,<breakpoint>] [,<watch_expression>]
k[ill] [<watch_expression>] {[,<breakpoint>] [,<watch_expression>]
```

Usage

`K[ill]` removes all specified <breakpoints> and <watch_expressions>. SrcDbg notation (i.e. b1, b2, w1, w2, etc.) is used to specify <breakpoints> and <watch_expressions>. For example:

```
kill b1, w1
```

When K[ill] is executed with no arguments, the following prompt is issued:

```
Kill all breakpoints and watch expressions? (y or n)
```

To remove all set breakpoints and watch expressions answer with a “y”. The same result may be accomplished with a wildcard:

```
SrcDbg: kill *
```

Wildcards may also be used to remove specifically only breakpoints or watch expressions in the following manner:

```
SrcDbg kill b*
SrcDbg kill w*
```

Examples

In the following example, where is forked with two parameters. These are passed to where as command line arguments. Three breakpoints are set. A B[reak] command is executed with no parameters to verify the breakpoints. Two breakpoints are removed with the K[ill] command. A B[reak] command without parameters displays the remaining breakpoint. Two watch expressions are set and displayed with the W[atch] command. All breakpoints and watch expressions are removed with the K[ill] command. Then, the B[reak] and W[atch] commands are entered without parameters to verify that all breakpoints and watch expressions have been removed.

```
$ srcdbg where where.c /h0/usr/kathie                                "where" forked
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: b where\where.c\35;b 91 :when *read_dir_stuff\$blk0\p == 's'        set 2
b1: where\where.c\35 :count 1                                             breakpoints
b2: where\where.c\91 :when *read_dir_stuff\$blk0\p == 's' :count 1
SrcDbg: break .pc+0x100                                         set another breakpoint
b3: where\fork_dir+0x70 :count 1
SrcDbg: break                                                   display all breakpoints
#  Breakpoint:                    Condition:                      Count:
-------------------------------------------------------------------------
 1 where\where.c\35                                               1
 2 where\where.c\91               *read_dir_stuff\$blk0\p == 's'  1
 3 where\fork_dir+0x70                                            1
SrcDbg: kill b1,b3                                          remove breakpoints 1 and 3
SrcDbg: break                                                   display all breakpoints
#  Breakpoint:                    Condition:                     Count:
------------------------------------------------------------------------
 2 where\where.c\91               *read_dir_stuff\$blk0\p == 's' 1
SrcDbg: watch *read_dir_stuff\$blk0\p                             set watch expression
w1: *read_dir_stuff\$blk0\p
SrcDbg: watch inbuf[10]                                   set another watch expression
w2: inbuf[10]
SrcDbg: watch                                             display all watch expressions
#  Watch Expression:
--------------------
 1 *read_dir_stuff\$blk0\p
 2 inbuf[10]
SrcDbg: kill *                          remove all breakpoints and watch expressions
SrcDbg: break;watch                     display all breakpoints and watch expressions
no break points
no watch expressions
SrcDbg:
```

Log

Write Commands to Logfile

Syntax

```
l[o]g <pathlist>
l[o]g : off
```

Usage

`L[o]g` writes SrcDbg commands to <pathlist>. The specified pathlist is relative to the user's current data directory. If ": off" is entered, the log file is closed.

The command file created by L[o]g may be read by the Re[ad] command.

Examples

In the following example, SrcDbg is forked with two parameters. The L[o]g command is used to open the file buglog and the subsequent commands are recorded in buglog . Then, the file is closed.

```
$ srcdbg where where.c /h0/usr/kathie                          fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: log /h0/usr/kathie/buglog                              create logfile "buglog"
SrcDbg: step                                                   command added to "buglog"
File: "where.c"
  19: void main(argc,argv)
           ^
SrcDbg(STEP): go fork_dir                                      command added to "buglog"
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
SrcDbg: watch fork_dir\argv[0]                                 command added to "buglog"
w1: fork_dir\argv[0]
SrcDbg: step                                                   command added to "buglog"
Watch Expression #1: fork_dir\argv[0]
0x2f94c = "where"
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)          ^
dn: FFFFFFA8 0002F890  00000003 00000003   0002F860 0002F85C  00001990 00000000
an: 0002F954 00000000  0002F954 0002F890   0002F864 0002F848  00036000 0002F824
pc:    E6A9E  cc:  08 (-N---)
fork_dir+0x1e       >6100FE32         bsr.w _stkchec
  51:   argv[0] = "dir";
        ^
SrcDbg(STEP): log : off                                        "buglog" closed
SrcDbg: list buglog                                            display the file "buglog"
   1: step                      (this file may now be used with the re[ad] command)
   2: go fork_dir
   3: watch fork_dir\argv[0]
   4: step
[End of file]
```

Option

Toggle Options

Syntax

```
o[ption] {<options>}
```

Usage

`O[ption]` allows the user to set a variety of display and execution options. The following options are currently available:

Option:	Default action (toggle action in parentheses):
fpu	Show (don't show) floating point coprocessor registers
fregs	Display floating point coprocessor registers in decimal (hexadecimal)
dbg	Read (don't read) ".dbg" symbols
stb	Read (don't read) ".stb" symbols
prompt	Show (don't show) prompt
echo	Don't echo (echo) command line output
rom	Show RAM hard (ROM soft) breakpoints
source	Show (don't show) source line with assembly locations
watch	Show (don't show) location of watch expression change

Examples

In the following example, SrcDbg is forked with two parameters and the O[ption] command is used to display the list of additional SrcDbg options. Then, the O[ption] command is used to set three options and the new status of the options is listed.

```
$ srcdbg where where.c /h0/usr/kathie                    fork “where”
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: option                                           display current option settings
Options:
         READ “dbg” symbols.
         READ “stb” symbols.
         SHOW floating point coprocessor registers.
         DECIMAL floating point coprocessor registers.
         SHOW location with watch expression changes.
         RAM (HARD) breakpoints.
         SHOW source with assembly level locations.
         NO ECHO
         PROMPT
SrcDbg: option fpu                                       toggle option “fpu”
SrcDbg: o fregs                                          toggle option “fregs”
SrcDbg: o dbg                                            toggle option “dbg”
SrcDbg: option                                           display current options
Options:
         DON’T READ “dbg” symbols.
         READ “stb” symbols.
         DON’T SHOW floating point coprocessor registers.
         HEXADECIMAL floating point coprocessor registers.
         SHOW location with watch expression changes.
         RAM (HARD) breakpoints.
         SHOW source with assembly level locations.
         NO ECHO
         PROMPT
SrcDbg:
```

Read

Read Commands From a File

Syntax

```
re[ad] [<pathlist>]
```

Usage

`Re[ad]` reads SrcDbg commands from <pathlist>. The <pathlist> is relative to the user's current data directory. The L[o]g command may be used to create the file referred to in <pathlist>.

Examples

In the following example, SrcDbg is forked with two parameters and the S[tep] command is used to move the execution pointer to main. Then, the Re[ad] command is used to read the commands from the file buglog. The `1.SrcDbg:` prompt signifies that the command interpreter has been re-entered to interpret commands.

```
$ srcdbg where where.c /h0/usr/kathie                    fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: read /h0/usr/kathie/buglog          read commands from file "buglog"
1.SrcDbg: step
File: "where.c"
  19: void main(argc,argv)
           ^
1.SrcDbg(STEP): go fork_dir
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
1.SrcDbg: watch fork_dir\argv[0]
w1: fork_dir\argv[0]
1.SrcDbg: step
  51:   argv[0] = "dir";
        ^
1.SrcDbg(STEP): quit
SrcDbg:
```

Chapter 8

Data Manipulation Commands

The commands described in this chapter control provide and manipulate program information for the user. They provide the following functions:

Command:	Description:
L[ist]	displays source listings of files
I[nfo]	returns information about a specified program object or your current location within the program
Fr[ame]	changes logical stack frame or displays stack frame information
P[rint]	prints the value of a specified C expression
A[ssign]	sets the value of a program object
C[h]c	changes the context for default name and location resolution
Con[text]	fully qualifies a symbol in terms of scope
Fi[nd]	displays all scope expressions found for <name>
Lo[cals]	displays the values of all local symbols

List

Display Source Code Listing

```
l[ist] <list_arg> [,<list_arg>]
```

L[ist] with no parameters displays the next 21 lines of source code beginning with the current line number. If the end of file occurs before 21 lines are displayed, an end-of-file message is displayed.

<list_arg> may be any of the following:

- Any OS-9 pathlist
- Any source file known to SrcDbg
- A scope expression resulting in a block number
- A scope expression resulting in a function
- A line number expression

If a <list_arg> is specified, L[ist] will then begin displaying source code at the beginning of the specified file or file section. L[ist] will display 21 lines of code even if the function or block is less than 21 lines (unless end-of-file is reached).

Two special cases exist for <list_args>. If the <list_arg> results in a line number, it may optionally be followed by a second line number. The source listing will begin at the first line number and end at the second. For example:

```
SrcDbg: list 1 2
   1: #include <stdio.h>
   2: #include <modes.h>
```

Specifying a beginning and ending line number allows the user to display as few or as many lines as personally desired. If a second line number is not specified, the default 21 line listing begins at the specified line number.

<list_args> may be separated by a comma on the L[ist] command line in order to display more than one portion of a file. For example:

```
SrcDbg: list 1 2, 19 20
   1: #include <stdio.h>
   2: #include <modes.h>
  19: void main(argc,argv)
  20: register int argc;
```

Examples

The following examples show various ways of using the L[ist] command. These examples assume the program where is being debugged:

```
SrcDbg: list                                   list from current position
  19: void main(argc,argv)
  20: register int      argc;
  21: char              *argv[];
  22: {
  23:   register char   *filename = argv[1];
  24:   auto int        pid,pipe,stout;
  25:
  26:   check_args(argc);
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
  28:   read_dir_stuff(filename,pid,pipe,stout);
  29: }
  30:
  31: void check_args(argc)
  32: register int argc;
  33: {
  34:   if(argc < 2) exit(_errmsg(0,“where <filename>
          [<pathlist>]\n”));
  35: }
  36:
  37: void fork_dir(argv,pathlist,pid,pipe,stout)
  38: register char       **argv,
  39:                     *pathlist;
SrcDbg(LIST): list 1 5                         list line 1 through 5
   1: #include <stdio.h>
   2: #include <modes.h>
   3: #include <errno.h>
   4: #define BUFSIZE 512
   5:
SrcDbg(LIST): list /h0/defs/errno.h            list an OS-9 file
   1: /* System call error numbers.
   2:  * May be found in ’errno’ after an error has occurred.
   3:  *
   4:  * These should be obtained from sys.l somehow.
   5:  *
   6:  */
   7:
   8: #define E_ILLFNC  0x40  /* (usually) Trap Error Codes        */
   9: #define E_FMTERR  0x41  /* illegal function code             */
  10: #define E_NOTNUM  0x42  /* number not found/not a number     */
  11: #define E_ILLARG  0x43  /* illegal argument                  */
  12:
  13: #define E_BUSERR  0x66  /* bus error TRAP 2 occurred         */
  14: #define E_ADRERR  0x67  /* address error TRAP 3 occurred     */
  15: #define E_ILLINS  0x68  /* illegal instruction TRAP 4 occur  */
  16: #define E_ZERDIV  0x69  /* zero divide TRAP 5 occurred       */
  17: #define E_CHK     0x6a  /* CHK instruction TRAP 6 occurred   */
  18: #define E_TRAPV   0x6b  /* TrapV instruction TRAP 7 occurred */
  19: #define E_VIOLAT  0x6c  /* privilege violation TRAP 8 occur  */
  20: #define E_TRACE   0x6d  /* Uninitialized Trace TRAP 9 occur  */
  21: #define E_1010    0x6e  /* Uninitialized 1010 TRAP 10 occur  */
```

list a specific block

```
SrcDbg(LIST): list where.c\main\$blk0
  23:   register char   *filename = argv[1];
  24:   auto int        pid,pipe,stout;
  25:
  26:   check_args(argc);
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
  28:   read_dir_stuff(filename,pid,pipe,stout);
  29: }
  30:
  31: void check_args(argc)
  32: register int argc;
  33: {
  34:   if(argc < 2) exit(_errmsg(0,"where <filename>
          [<pathlist>]\n"));
  35: }
  36:
  37: void fork_dir(argv,pathlist,pid,pipe,stout)
  38: register char       **argv,
  39:                     *pathlist;
  40: register int        *pid,
  41:                     *pipe,
  42:                     *stout;
  43: {
```

list a specific function

```
SrcDbg(LIST): list check_args
  32: void check_args(argc)
  33: register int argc;
  34: {
  35:   if(argc < 2) exit(_errmsg(0,"where <filename>
          [<pathlist>]\n"));
  36: }
  37:
  38:
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
  40: register char     **argv,
  41:                   *pathlist;
  42: register int      *pid,
  43:                   *pipe,
  44:                   *stout;
  45: {
  46:   register int    stin;
  47:   extern int      os9fork(), os9exec();
  48:   extern char     **environ;
  49:
  50:   /* initialize argv list for dir */
  51:   argv[0] = "dir";
  52:   argv[1] = "-rasu";
```

list lines 20-24 and 32-33

```
SrcDbg(LIST): list 20 22,32 33
  20: register int      argc;
  21: char              *argv[];
  22: {
  32: void check_args(argc)
  33: register int argc;
```

Info

Start Program

Syntax

```
i[nfo] <scope_expression>
```

Usage

I[nfo] returns information about specified program objects and the current program location. When I[nfo] is given with no parameters, SrcDbg displays the current location of the program. For example:

```
SrcDbg: info
File: "where.c"
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg:
```

<scope_expression> specifies a program object using a scope expression. For a detailed discussion of scope expressions, see the discussion on command syntax in the Overview to SrcDbg chapter.

When a function is specified by the I[nfo] command, a declaration of the function data type is displayed. Then declarations of each of the formal parameters "expected" data types are displayed. For example:

```
SrcDbg: info read_dir_stuff
void read_dir_stuff(filename,pid,pipe,stout)
register char *filename;
register int pid;
register int pipe;
register int stout;
```

When a struct, enum or union type is specified by an I[nfo] command, all typedef declarations using the specified type are declared. For example:

```
SrcDbg: info FILE
typedef struct _iobuf FILE;
```

When a struct, enum or union tag is specified by an I[nfo] command, the full declaration is displayed. For example:

```
SrcDbg: info struct _iobuf
struct _iobuf {
    char *_ptr;
    char *_base;
    char *_end;
    WORD _flag;
    WORD _fd;
    char _save;
    WORD _bufsiz;
    int (*_ifunc)();
    int (*_ofunc)();
};
```

`I[nfo]` also returns information on a variety of other items:

- Basic C types (includes modules, registers, etc.)
- files (includes files, source files, etc.)
- block numbers
- registers

For example:

```
SrcDbg: info int
C Basic Type int
SrcDbg: info where.c
Source File "where.c"
SrcDbg: info /dd/defs/modes.h
Include File "/dd/defs/modes.h"
SrcDbg: info where
Program "where"
SrcDbg: info $blk0
Compound Statement $blk0
SrcDbg: info .d0
register long .d0;
SrcDbg: info .fp0
register double .fp0;
```

If no information is available about a specified object, SrcDbg returns an appropriate message. For example:

```
SrcDbg: info $blk21
"$blk21" not found
```

Examples

The following examples show further uses of the I[nfo] command using the where program:

```
$ srcdbg where where.c /h0/usr/kathie                    fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: step                                             step to main
File: "where.c"
  19: void main(argc,argv)
           ^
SrcDbg: info boolean                                     return information on "boolean"
typedef enum {
    FALSE = 0,
    TRUE = 1,
} boolean;
srcdbg: info sigflag                                     return information on "sigflag"
boolean sigflag;
SrcDbg: info .pc                                         return information on ".pc" register
register unsigned long .pc;
```

Frame

Display Stack Frame Information

Syntax

```
f[rame] [[+] <number>]
```

Usage

F[rame] displays stack frame information. If no arguments are specified, the name of each calling function, the location from which it was called and the frame number are displayed. The active frame is designated with a “*”.

If F[rame] is followed by <number>, the stack frame context will be changed to <number>. This enables the user to access all variable values local to the specified frame. If F[rame] is followed by a signed number, the stack frame will be changed relative to the current frame +/– <number> .

The F[rame] command does not effect the location of the execution pointer.

F[rame] also displays the values of each of the actual parameters passed to the calling function with the function’s formal parameter names.

At the left of each calling function is the frame number. Frame number zero is the current frame. Each frame is successively numbered (1, 2, 3, etc.).

Important: If a frame for which there is no source information is referenced or “passed through”, the following warning will be displayed:

```
Warning - register variables may not have correct values.
[stack frame not intact]
```

The following example forks the program where. Two parameters are passed to the program. A breakpoint is set at the function check_args. where is run to the breakpoint using the G[o] command. A F[rame] command is executed to show the frame stack. Two different F[rame] commands are executed to change the frame stack context to 1 and display the new frame stack. A R[eturn] command is executed to return one frame. The F[rame] command is again executed to show the result of the previous command:

```
$ srcdbg where where.c /h0/usr/kathie          “where” forked with 2 parameters
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: step
File: “where.c”
  19: void main(argc,argv)
           ^
SrcDbg(STEP): break check_args                                set breakpoint
b1: where\check_args+0xa :count 1
SrcDbg: go   run to breakpoint
At breakpoint #1
File: “where.c”
Context: where.c\check_args
  32: void check_args(argc)
           ^
SrcDbg: frame                                                 show frame stack
  #  Location of Call:      Call:
-------------------------------------------------------
* 0: where.c\26              check_args(argc = 3)
  1: _cstart+0xe4            main(argc = 3, argv = 0x29952)
SrcDbg: frame 1                               change frame stack context to 1
SrcDbg: frame                                                 show frame stack
  #  Location of Call:       Call:
--------------------------------------------------------
  0: where.c\26              check_args(argc = 3)
* 1: _cstart+0xe4            main(argc = 3, argv = 0x29952)
SrcDbg: print argv[1]                          display contents of argv[1]
0x29888 = “where.c”
SrcDbg: return 1                                              return one frame
File: “where.c”
Context: where\main\$blk0
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
        ^
SrcDbg: frame                                                 show frame stack
  #  Location of Call:       Call:
--------------------------------------------------------
* 0: _cstart+0xe4            main(argc = 3, argv = 0x30954)
SrcDbg:
```

Print

Print Expression Value

Syntax

```
P[rint] [<C_expr>]
```

Usage

`P[rint]` returns the value of the specified <C_expr>. `P[rint]` displays the value according to the resulting data type of the expression. All program objects referred to within the <C_expression> are relative to the current scope, with one exception: "static" variables outside the current scope may be accessed using scope expressions.

The printed values are displayed in the following format:

Parameter type:	Display format type:
char pointer	<hex address> = <string>
all other pointers	<hex address>
array	array elements in correct type
struct	struct elements in correct type
union	union fields in correct type
enum	enum constant name of the value or Decimal if no name has the actual value
char	Character constant
short	Decimal
int	Decimal
long	Decimal
unsigned	Decimal
unsigned char	Decimal
unsigned short	Decimal
unsigned int	Decimal
unsigned long	Decimal
float	float
double	float

Important: SrcDbg will prompt for the number of elements to be printed if a unsized array is specified. SrcDbg will display "union", if a union is specified. To obtain a useful value, a union field must be specified.

To change the format of the P[rint] output, it is necessary to cast the type of the program object be printed to the desirable type. For example:

```
SrcDbg: print argc
2
SrcDbg: print (void *) argc
0x2
```

The P[rint] command can accept a function as a parameter. For example:

```
SrcDbg: print main()
```

SrcDbg will run the function and display the function's returning value. If a breakpoint is set within the function, SrcDbg will stop at the breakpoint. A new prompt will be issued showing that SrcDbg has stopped while executing a previous command. For example:

```
1.SrcDbg:
```

When SrcDbg exits from the function, the normal SrcDbg prompt is resumed and the functions returning value will be printed.

If only the function name is specified as a P[rint] parameter, only the address for the function is returned. For example:

```
SrcDbg: print main
0x240846
```

Examples

The following examples show various uses of the P[rint] command. where is forked with 2 parameters. The print command is used to show each of the parameters passed to where. The program is executed up to a breakpoint set at line 83. The P[rint] command is used again to show the search directory:

```
$ srcdbg where where.c /h0/usr/kathie                  "where" forked with 2 parameters
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: go main
File: "where.c"
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg: step
  23:   register char   *filename = argv[1];
     ^
SrcDbg(STEP): print argc                                                print argc
3
SrcDbg: print argv                                                      print argv
0x27954
```

```
SrcDbg: print argv[0]                                            print argv[0]
0x2794c = “where”
SrcDbg: print argv[1]                                            print argv[1]
0x27888 = “where.c”
SrcDbg: print argv[2]                                            print argv[2]
0x27890 = “/h0/usr/kathie”
SrcDbg: print *filename                     print first character of filename
’\0’
SrcDbg: print filename                                  print filename string
0x27958 = “”
SrcDbg: go 86                                            execute to line 86
File: “where.c”
Context: where\read_dir_stuff\$blk0  86:   while(gets(buf)) {
              ^
SrcDbg: print buf                                                  print buf
0x2601e = “”
SrcDbg: s
  87:       if(str = (char *)rindex(buf,’/’)) {
               ^
SrcDbg(STEP): print buf                             print buf; note new value
0x2601e = “/h0/usr/kathie/SEA”
SrcDbg: print (char *) inbuf                      print inbuf; note same value
0x2601e = “/h0/usr/kathie/SEA”
SrcDbg: print inbuf                       print array contents until [Ctrl-C]
’/’, ’h’, ’0’, ’/’, ’u’, ’s’, ’r’, ’/’, ’k’, ’a’, ’t’, ’h’, ’i’, ’e’, ’/’, ’S’,
’E’, ’A’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’,
’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’, ’\0’,
’\0’, ’\0’,  [aborted]
SrcDbg: s 3
  88:           *str = ’\0’;
                ^
  89:           p = str + 1;
                ^
  91:       if(!strcmp(p,filename)) puts(buf);
               ^
SrcDbg(STEP): print filename                                  print filename
0x27888 = “where.c”
SrcDbg: print p                                                      print p
0x2602d = “SEA”
SrcDbg: s
  92:       if(sigflag) break;
               ^
SrcDbg(STEP): s
  86:   while(gets(buf)) {
              ^
SrcDbg(STEP): print sigflag                                    print sigflag
FALSE
SrcDbg: print .r0                                         print  register .r0
0
SrcDbg: print (char *) .sr                              print status register
0x8014
SrcDbg: g
/h0/usr/kathie/SEA                                              where output
/h0/usr/kathie/SOURCE
“where” exited normally
SrcDbg:
```

Assign

Assign Value to Expressionm

Syntax

```
a[ssign] [<C_expr>] = [<C_expr>]
```

Usage

`A[ssign]` sets the value of a program object.

Examples

The following example forks where. Two parameters are passed to the program. The P[rint] command is used to show the value of argc. The value of argc is then changed using the A[ssign] command. The same sequence is then executed with the variable argv[1]:

```
$ srcdbg where where.c /h0/usr/kathie          "where" forked with 2 parameters
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: go main
File: "where.c"
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg: print argc                             print value of argc
3
SrcDbg: assign argc = 50000000                 new value for argc
SrcDbg: print argc
50000000
SrcDbg: print argv[1]                          print value for argv[1]
0x27888 = "where.c"
SrcDbg: assign *(argv[1] + 5) = '\0'           new value for argv[1]
SrcDbg: print argv[1]
0x27888 = "where"
SrcDbg: assign .r1 = argv[1]                   save expression in .r1
SrcDbg: print (char *) .r1                     print value of .r1
0x29888 = "where.c"
SrcDbg:
```

Chc

Change Context

Syntax

```
c[h]c [<scope_expr>]
```

Usage

`C[h]c` changes the context for default name and location resolution. If the parent of the block specified in <scope_expr> is in the stackframe, then the logical context is changed to that frame number. If the parent of the block is not in the frame stack, then the logical context is changed to the specified block and no variables will be active.

The context may also be changed to a file or module. If the <scope_expr> is a file, the context will be the first function in the file. If the <scope_expr> is a module, the context will be the first function of the first file in the module.

If no argument is specified, the C[h]c command returns SrcDbg to the location of the execution pointer.

The C[h]c command, like the F[rame] command, does not effect the location of the execution pointer.

Examples

In the following example, SrcDbg is forked with two parameters and the execution pointer is moved main. The C[h]c command is used to change the default context to the function fork_dir. The L[ist] command is used to list the lines 39 - 59 and the I[nfo] command is used to display the logical and actual locations in the program. The S[tep] command is used to advance the execution pointer one statement past main. The C[h]c command is used to change context to the function check_args, and the I[nfo] command is used to obtain information about the variable argc. Then, the G[o] command is used to run where to completion.

```
$ srcdbg where where.c /h0/usr/kathie                    fork where
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: go main                         move execution pointer to main
File: “where.c”
Context: where\main
  19: void main(argc,argv)
           ^
```

```
SrcDbg: chc fork_dir                                      change context  to fork_dir
SrcDbg: list                                                     list lines 39-59
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
  40: register char     **argv,
  41:                   *pathlist;
  42: register int      *pid,
  43:                   *pipe,
  44:                   *stout;
  45: {
  46:   register int    stin;
  47:   extern int      os9fork(), os9exec();
  48:   extern char     **environ;
  49:
  50:   /* initialize argv list for dir */
  51:   argv[0] = "dir";
  52:   argv[1] = "-rasu";
  53:   argv[2] = pathlist;
  54:   argv[3] = NULL;
  55:
  56:   /* dup standard out "save it" and close standard out */
  57:   *stout = dup(1);
  58:   close(1);
  59:
SrcDbg(LIST): i                            get information about program location
Logical Location:
File: "where.c"
Context: where\fork_dir
  39: void fork_dir(argv,pathlist,pid,pipe,stout)
           ^
Actual Location:
File: "where.c"
Context: where\main
  19: void main(argc,argv)
SrcDbg(LIST): step                                           step to line after main
  23:   register char   *filename = argv[1];
                                         ^
SrcDbg(STEP): cc check_args                          change context to check_args
SrcDbg: i argc                                          get information about argc
register int argc;
SrcDbg(LIST): go                                         run where to completion
SrcDbg: g
/h0/usr/kathie/SEA                                                  where output
/h0/usr/kathie/SOURCE
"where" exited normally
SrcDbg:
```

Context

Displays Scope Expression

Syntax

```
con[text] [<scope_expression>]
```

Usage

`Con[text]` displays the complete scope expression of an object.

Examples

In the following example, where is forked with two parameters and the execution pointer is moved to main. The Con[text] command is used to obtain a full scope expression for the variable argc.

```
$ srcdbg where where.c /h0/usr/kathie                    fork “where”
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: go main                         move execution pointer  to main
File: “where.c”
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg: con argc                          display the context of argc
where\main\argc
SrcDbg:
```

Find

Displays All Occurrences of <name>

Syntax

```
fi[nd] [<name>]
```

Usage

`Fi[nd]` displays all scope expressions for <name> found in the debug process.

If Fi[nd] is entered without any arguments, the previous Fi[nd] command will be repeated. If there was no previous Fi[nd] command, an error message will be returned:

```
Error: no previous “find”
```

Examples

In this example, file (page 8 in Overview of SrcDbg) is forked. The Fi[nd] command is used to locate every instance of the variable 'i' in file.

```
$ srcdbg file                                    fork “file”
Reading symbol file “file.dbg”.
file1.c             file2.c
Reading symbol file “file.stb”.
Context: file\_cstart
SrcDbg: find i                                   find every instance of the
file\main\$blk0\$blk1\i                           variable 'i' in file
file\file2.c\f\$blk0\i
file\main\$blk0\i
file\file2.c\struct tag1\i
file\f\i
SrcDbg:
```

Locals

Display Local Symbol Values

Syntax

```
lo[cals]
```

Usage

`Lo[cals]` displays the values of all local symbols.

Examples

In the following example, where is forked with two parameters and the S[tep] command is used to move the execution pointer to line 26. Then, the Lo[cals] command is used to obtain the values of all the local variables.

```
$ srcdbg where where.c /h0/usr/kathie                fork where
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: step 3                                       step to line 26
File: "where.c"
  19: void main(argc,argv)
           ^
  23:   register char   *filename = argv[1];
                                              ^
  26:   check_args(argc);
        ^
SrcDbg(STEP): locals                                 get values on
filename = 0x27888 = "where.c"                       all local variables
pid = 944634
pipe = 158848
stout = -1592735107
argc = 3
argv = 0x27954
SrcDbg:
```

Chapter 9

System Commands

The following commands are described in this chapter:

Command:	Description:
C[h]d	changes the data directory SrcDbg uses to search for files
Sh[ell]	forks a new Shell to allow access to OS-9
H[elp]	provides a help for SrcDbg commands
Q[uit]	exits SrcDbg and returns command to the Shell
C[h]x	changes the current execution directory for SrcDbg
Se[tenv]	sets a shell-type environment variable
Unse[tenv]	deletes a shell-type environment variable

Chd

Change Directory

Syntax

```
C[h]d <pathlist>
```

Usage

`C[h]d` changes SrcDbg's current data directory. This does not change the current directory of the parent Shell (i.e. the current directory when SrcDbg was invoked).

Examples

The following example shows how the C[h]d command can be used. The program where is forked, but the file where.c is not found. The C[h]d command is used to move to the correct directory in order to fork the program:

```
$ srcdbg where where.c /h0/usr/kathie                         "where" forked
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: step
File: "where.c"
Could not open file "where.c" - Error #000:216(E$PNNF)File not found.
SrcDbg(STEP): chd /h0/usr/kathie/source                     Change directory
SrcDbg: info                                           display source location
File: "where.c"
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg: step
  23:   register char   *filename = argv[1];
                                           ^
SrcDbg(STEP):
```

Shell

Fork Shell

Syntax

```
Sh[ell] [<command>]
$ [<command>]
```

Usage

`Shell` executes any specified Shell <command> and returns control to SrcDbg. If no <command> is specified, the user is placed in a new Shell. To return to SrcDbg, exit the shell using the logout command or **[Esc]**.

Important: If a complex Shell command is desired using a semi-colon (";"), a Shell should be forked by itself. Otherwise, the semicolon will be interpreted by SrcDbg, not the Shell. For example, in the following SrcDbg command, the semicolon separates a Shell command and a SrcDbg command:

```
SrcDbg: shell dir;info argc
```

Examples

The following examples show the use of the S[hell] command:

```
SrcDbg: shell dir                          execute dir command and return to SrcDbg
                 Directory of . 16:40:32
capture         capture.new     capture2        junk            mbox
where.c         where.list
SrcDbg: shell                                                   fork new shell
1.walden: dir
                 Directory of . 16:40:32
capture         capture.new     capture2        junk            mbox
where.c         where.list
1.walden: logout                              exit Shell and return to SrcDbg
SrcDbg: $ dir -u * ! grep -nz pathlist                       grep for pathlist
filename="where.c"
   12     fork_dir(argv,pathlist,pid,pipe,stout),
   35   if(argc<2) exit(_errmsg(0,"where <filename>
[<pathlist>]\n"));
   39 void fork_dir(argv,pathlist,pid,pipe,stout)
   41                  *pathlist;
   53   argv[2] = pathlist;
SrcDbg: $                                                       fork new shell
1.walden:
```

Help

Display SrcDbg Help

Syntax

```
H[elp]
```

```
?
```

Usage

SrcDbg provides on-line help when H[elp] or ? is entered.

Quit

Exit SrcDbg

Syntax

```
q[uit]
```

```
<escape>
```

Usage

`Q[uit]` exits SrcDbg or aborts an interrupted function call. If SrcDbg is exited, all open files are closed and control returns to the Shell. If a interrupted function call is aborted, SrcDbg returns to the debugging session at the point from which the function call was made.

Chx

Change Execution Directory

Syntax

```
c[h]x <pathlist>
```

Usage

`C[h]x` changes the current execution directory for SrcDbg. This command effects where SrcDbg will look for program object, ".dbg" and ".stb" files.

`C[h]x` does not change the current execution directory of the parent Shell (i.e. the current directory when SrcDbg was invoked).

Examples

The following example shows the use of the C[h]x command:

```
$ srcdbg where where.c /h0/usr/kathie            attempt to fork "where" ; fail
Could not fork "where" - Error #000:216 (E$PNNF) File not found.
SrcDbg: chx /h0/cmds/kathie                      change execution directory
SrcDbg: fork where where.c /h0/usr/kathie        attempt to fork "where"
Reading symbol file "where.dbg".                 success!
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg:
```

Setenv

Set Environment Variable

Syntax

```
se[tenv] <environment_name> <environment_definition>
```

Usage

`Se[tenv]` sets a shell-type environment variable for use by SrcDbg and SrcDbg's child processes. The arguments <environment_name> and <environment_definition> are strings that are stored in the environment list by SrcDbg.

`Se[tenv]` does not change the environment of the parent Shell (i.e. the shell from which SrcDbg was invoked).

Examples

The following example shows how the Se[tenv] command is used:

```
$ srcdbg where where.c /h0/usr/kathie                    "where" forked
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: $ printenv                          display environment variables
PORT=/t21
HOME=/h0/USR/KATHIE
SHELL=shell
USER=kathie
PATH=..
TERM=kt7
SrcDbg: setenv PATH ../h0/cmds:/d0/cmds:/dd/cmds               sets PATH
SrcDbg: $ printenv                          display environment variables
PORT=/t21
HOME=/h0/USR/KATHIE
SHELL=shell
USER=kathie
PATH=../h0/cmds:/d0/cmds:/dd/cmds
TERM=kt7
SrcDbg:
```

Unsetenv

Unset Environment Variable

Syntax

```
unse[tenv] <environment_name>
```

Usage

`Unse[tenv]` deletes an environment variable from the environment list.

`Unse[tenv]` does not change the environment of the parent Shell (i.e. the shell from which SrcDbg was invoked).

Important: If the specified <environment_name> has not been previously defined, Unse[tenv] has no effect and does not print a message.

Examples

The following example shows how the Unse[tenv] variable is used:

```
$ srcdbg where where.c /h0/usr/kathie                    "where" forked
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: $ printenv                                       display environment variables
PORT=/t21
HOME=/h0/USR/KATHIE
SHELL=shell
USER=kathie
PATH=../h0/cmds:/d0/cmds:/dd/cmds
TERM=kt7
SrcDbg: setenv BUGS bunny                                set environment variable BUGS
SrcDbg: $ printenv                                       display environment variables
PORT=/t21
HOME=/h0/USR/KATHIE
SHELL=shell
USER=kathie
PATH=../h0/cmds:/d0/cmds:/dd/cmds
TERM=kt7
SrcDbg: unsetenv BUGS                                    unset BUGS
SrcDbg: $ printenv                                       display environment variables
PORT=/t21
HOME=/h0/USR/KATHIE
SHELL=shell
USER=kathie
PATH=../h0/cmds:/d0/cmds:/dd/cmds
TERM=kt7
SrcDbg:
```

Chapter 10

Assembly Level Commands

The following commands are described in this chapter after a discussion of assembly level display information:

Command:	Description:
Asm	displays current register values and the current machine instruction
C[hange]	changes memory at the specified location
Dil[ist]	displays C source with disassembly at the specified location
Di[sasm]	disassembles memory at the specified location
D[ump]	displays memory at the specified location
Gostop	executes a number of machine instructions in the current subroutine
Li[nk]	links to a module
Mf[ill]	fills memory with the specified value
Ms[earch]	searches memory for the specified value
Sy[mbol]	displays the result of the expression as a symbolic expression
T[race]	executes a number of machine instructions

Assembly Level Display Information

The register display appears as follows:

```
dn: 0000000D 0000004C  00000080 00000003   00000000 00000108  00001990 00000000
an: 00000000 0002F990  00000000 000E6780   00000000 0002F888  00036000 0002F888
pc:    E67CE  cc:  00 (-----)
_cstart             >2D468010         move.l d6,_totmem(a6)
```

The first two lines of the display show the data and address registers from .d0 - .d7. and .a0 - .a7 respectively. The third line shows the program counter and status register. Only the user byte (containing the processor condition code values) are available. The condition code bits are interpreted and displayed after the condition code register hex value in parentheses.

The bit interpretation is:

```
(XNZVC)

X = Extend
N = Negative
```

```
Z = Zero
V = Overflow
C = Carry
```

The following discussion of the 68881 coprocessor registers applies only to OS-9 systems running on a 68020 processor with the 68881 installed as a coprocessor.

If the display 68881 registers option is set, the next line(s) indicate the state of the 68881 coprocessor. If the process has yet to access the 68881, the following message appears:

```
<68881 in Null state>
```

When the process accesses the 68881, a floating point register dump will appear. If a decimal display is indicated (SrcDbg defaults to this), the registers appear in the following format:

```
fp0:6.666666666666666          fp4:<NaN>                         fpcr: 0000 XN
fp1:<NaN>                      fp5:<NaN>                        fpiar: 00000000
fp2:<NaN>                      fp6:<NaN>                         fpsr: 00000208
fp3:<NaN>                      fp7:<NaN>                         (----    0)
```

If the setting of the debugger decimal register display option indicates hex display, the 68881 coprocessor registers appear as follows:

```
fp0:40010000 D5555555 55555200  fp4:7FFF0000 FFFFFFFF FFFFFFFF   fpcr: 0000 XN
fp1:7FFF0000 FFFFFFFF FFFFFFFF  fp5:7FFF0000 FFFFFFFF FFFFFFFF  fpiar: 00000000
fp2:7FFF0000 FFFFFFFF FFFFFFFF  fp6:7FFF0000 FFFFFFFF FFFFFFFF   fpsr: 00000208
fp3:7FFF0000 FFFFFFFF FFFFFFFF  fp7:7FFF0000 FFFFFFFF FFFFFFFF   (----    0)
```

The value of the registers are printed in decimal using scientific notation when the value becomes very large or very small. IEEE not-a-number values are printed as <NaN>, plus and minus infinity values are printed as <+Inf> and <-Inf>, respectively. The extended precision values are converted to double precision before printing, so conversion overflow may result. The hexadecimal format display can be used to determine the exact values in the registers.

The eight 68881 floating point registers are displayed in either hex or decimal form depending on the floating point register display option setting.

The 68881 status registers appear to the far right of the display:

`fpcr: 0000 --`	68881 control register
`fpiar: 00000000`	68881 instruction address register

```
fpsr: 00000000              68881 status register
(----    0)                 FPSR interpretation bits
```

The — field next to the FPCR register displays an interpretation of the 68881 rounding mode and precision. These fields are interpreted as follows:

```
fpcr: 0000 --
            ↑↑
            | Rounding mode:     N = Nearest
            |                    Z = Toward zero
            |                    – = Toward minus infinity
            |                    + = Toward plus infinity
            |
           Rounding Precision:   X = Extended
                                 S = Single
                                 D = Double
                                 ? = Undefined
```

The FPSR condition code byte and the quotient byte are displayed as follows:

```
   (----    0)
↑↑↑↑        ↑
||||        Quotient byte value (displays in signed decimal)
||||
||||        Floating point condition codes:
|||
||| ? = NaN or Unordered
||  I = Infinity
|   Z = Zero
    N = Negative
```

Immediately following the main floating register display, the debugger will interpret the exception enable byte of the control register and the exception status and accrued exception bytes of the status register. If all bits in the byte are zero, nothing is printed. Otherwise the bits are displayed as follows:

```
XE:(BSUN,SNAN,OPERR,OVFL,UNFL,DZ,INEX2,INEX1)   FPCR exception enable
AX:(IOP,OVFL,UNFL,DZ,INEX,???,???,???)          FPSR accrued exception
XS:(BSUN,SNAN,OPERR,OVFL,UNFL,DZ,INEX2,INEX1)   FPSR exception status
```

Processor registers can be changed with the A[ssign] command. Any processor register or coprocessor control register can be changed with this command:

```
SrcDbg: a .d0 = 0
SrcDbg: a .fp0 = 1.0
```

The following registers are supported for the MC68xxx:

Register:	Description:
.d0 - .d7.	data registers
.a0 - .a7	address registers (.sp may be used for .a7)
.sr	lower order byte of the status register (.cc may be used for .sr)
.pc	program counter

These registers are supported for the MC68881 floating point coprocessor:

Register:	Description:
.fp0 - .fp7	floating point registers
.fpcr	control register
.fpsr	status register
.fpiar	instruction address register

SrcDbg also supports the use of debugger convenience registers .r0 - .r7. These registers are available so that the user may store expression results for later access.

Instruction Disassembly Memory Display

The instruction disassembly display format, conditional instructions may be followed with a –> indicator. If –> is present, the instruction will perform its TRUE operation, otherwise the instruction performs the FALSE operation. The appropriate condition code register is examined to determine which case the processor will perform. The following conditional instruction categories use this feature:

Category:	Description:
Bcc	Branch on condition
DBcc	Decrement and branch on condition
Scc	Set According to condition
TRAPcc	Trap on condition
FBcc	Branch on floating condition
FScc	Set According to floating condition
FDBcc	Decrement and branch on floating cond
FTRAPcc	Trap on floating condition

To display a floating point number in machine registers, use the print command. For example:

Example:

instruction disassembly

```
SrcDbg: di _cstart
_cstart             >2D468010          move.l d6,_totmem(a6)
_cstart+0x4         >2D468014          move.l d6,_sbsize(a6)
_cstart+0x8         >3D438018          move.w d3,_pathcnt(a6)
_cstart+0xC         >4A85              tst.l d5
_cstart+0xE         >671E              beq.b _cstart+0x2E
_cstart+0x10        >08050000          btst.b #0,d5
_cstart+0x14        >6614              bne.b _cstart+0x2A->
_cstart+0x16        >41F55800          lea.l 0(a5,d5.l),a0
_cstart+0x1A        >4A68FFFE          tst.w -2(a0)
_cstart+0x1E        >660A              bne.b _cstart+0x2A->
_cstart+0x20        >5988              subq.l #4,a0
_cstart+0x22        >49E8FFFC          lea.l -4(a0),a4
_cstart+0x26        >7001              moveq.l #1,d0
_cstart+0x28        >6020              bra.b _cstart+0x4A
_cstart+0x2A        >423558FF          clr.b -1(a5,d5.l)
_cstart+0x2E        >204D              movea.l a5,a0
```

disassemble 5 instructions

```
SrcDbg(DISASM): di _cstart 5
_cstart             >2D468010          move.l d6,_totmem(a6)
_cstart+0x4         >2D468014          move.l d6,_sbsize(a6)
_cstart+0x8         >3D438018          move.w d3,_pathcnt(a6)
_cstart+0xC         >4A85              tst.l d5
_cstart+0xE         >671E              beq.b _cstart+0x2E
```

Floating Point Memory Displays

The floating point conditional instructions use the condition portion of the 68881 FPSR register, the others use the processor CC register. During memory disassembly display the –> indicator appears based on the static value of the condition register value when the disassembly occurred. The following are examples of floating point memory displays:

```
SrcDbg: dump f_ :1                                   display in hex/ascii format
f_                    - 3F2AAAAB 00000000 00000000 00000000  ?**+............
SrcDbg(DUMP): dump f_ :float                         display in single-precision decimal
f_                    - 3F2AAAAB 0.6666666865348816
SrcDbg(DUMP): dump a_ :1                             display in hex/ascii format
a_                    - 3FE55555 55555555 3F2AAAAB 00000000  ?eUUUUUU?**+....
SrcDbg(DUMP): dump a_ :double                        display in double–precision decimal
a_                    - 3FE5555555555555 0.6666666666666666
```

To display a floating point number in machine registers, use the print command. For example:

```
SrcDbg: print .fp0
2.530733e-07
```

Asm

Display Current Register Values and Machine Instruction

Syntax

```
asm
.
```

Usage

`Asm` displays current register values and the current machine instruction.

Examples

The following example uses the Asm command to display the register values at where_cstart:

```
$ srcdbg where where.c /h0/usr/kathie
Reading symbol file “where.dbg”.
where.c
Reading symbol file “where.stb”.
Context: where\_cstart
SrcDbg: .
dn: 0000000D 0000004C  00000080 00000003   00000000 00000108  00001990 00000000
an: 00000000 0002F990  00000000 000E6780   00000000 0002F888  00036000 0002F888
pc:    E67CE  cc:  00 (-----)
_cstart             >2D468010         move.l d6,_totmem(a6)
SrcDbg:
```

Change

Change Memory

Syntax

```
c[hange] [<C_expr>]                  change byte
cw [<C_expr>]                        change word
changeword [<C_expr>]                change word
cl [<C_expr>]                        change longword
changelong [<C_expr>]                change longword
```

Usage

`C[hange]` changes memory starting at the result of <C_expr>.

When the C[hange] command is invoked, SrcDbg displays the first byte/word/longword at the location specified by <C_expr> and then waits for the user to respond.

If a C expression is entered at this point, the memory at that location will be changed to the result of the entered expression.

If a "–" sign is entered, SrcDbg moves back one byte/word/longword without changing memory contents. If a **`[Return]`** is entered, SrcDbg moves forward one byte/word/longword without changing memory contents.

If a period is entered, the C[hange] command is terminated and the user is returned to a "`SrcDbg:` " prompt.

Examples

The following example shows how the C[hange] command is used to alter the memory at inbuf. The user responses are shown in italics.

```
$ srcdbg where where.c /h0/usr/kathie                    fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: change inbuf        change/examine bytes of memory beginning at "inbuf"
inbuf          : 00 0xff                                 enter a hex number
inbuf+0x1      : 00 'c'                            enter a character constant
inbuf+0x2      : 00 12                               enter a decimal number
inbuf+0x3      : 00 *argv[0]                           enter a C expression
inbuf+0x4      : 00 -                                   move back one byte
inbuf+0x3      : 77 <return>                         move forward one byte
inbuf+0x4      : 00 .                                       exit C[hange]
SrcDbg: change inbuf        change/examine bytes of memory beginning at "inbuf"
inbuf          : FF <return>                   successive carriage returns
inbuf+0x1      : 63 <return>                      display contents of "inbuf"
inbuf+0x2      : 0C <return>
inbuf+0x3      : 77 <return>
inbuf+0x4      : 00 <return>
inbuf+0x5      : 00 <return>
inbuf+0x6      : 00 .                                       exit C[hange]
SrcDbg: changelong inbuf  change/examine longwords of memory beginning at "inbuf"
inbuf          : FF630C77 0                          enter a decimal number
inbuf+0x4      : 00000000 .                                 exit C[hange]
SrcDbg: changeword          change/examine words of memory beginning at "inbuf"
inbuf          : 0000 argc                             enter a C expression
inbuf+0x2      : 0000 -                                 move back one word
inbuf          : 0003 .                                     exit C[hange]
SrcDbg:
```

Dilist

Display C Source With Disassembly

Syntax

```
dil[ist] [<location_expr>][: <count>]
```

Usage

`Dil[ist]` displays the current C source line and the assembly code which maps to the current C source line starting at the address specified by <dil_expr>. If <count> is specified, then <count> lines will be displayed.

If Dil[ist] is entered without any arguments, the previous Dil[ist] command will be repeated. If there was no previous Dil[ist] command, an error message will be returned:

```
Error: no previous “dilist”
```

After a Dil[ist] command has been executed, SrcDbg displays the following prompt:

```
SrcDbg(DILIST):
```

To continue “dilisting” the contents following the last address displayed by the previous Dil[ist] command, enter **[Return]**.

Example

The following example shows how the Dil[ist] command is used:

```
$ srcdbg where where.c /h0/usr/kathie                              fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: dilist main:10                       display C source line and disassembly
  19: void main(argc,argv)                   for 10 lines starting at function main
           ^
main+0xa            >203CFFFFFFA4     move.l #-92,d0
main+0x10           >6100FEDA         bsr.w _stkchec
main+0x14           >4FEFFFF4         lea.l -12(a7),a7
  23:   register char   *filename = argv[1];
                                           ^
main+0x18           >206F0010         movea.l 16(a7),a0
main+0x1c           >24680004         movea.l 4(a0),a2
  26:   check_args(argc);
        ^
main+0x20           >2004             move.l d4,d0
main+0x22           >61000040         bsr.w check_args
SrcDbg(DILIST): dilist 26 :10                display C source line and disassembly
  26:   check_args(argc);                    for 10 lines starting at C source line #26
        ^
main+0x20           >2004             move.l d4,d0
main+0x22           >61000040         bsr.w check_args
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
        ^
main+0x26           >4857             pea.l (a7)
main+0x28           >486F0008         pea.l 8(a7)
main+0x2c           >486F0010         pea.l 16(a7)
main+0x30           >206F001C         movea.l 28(a7),a0
main+0x34           >22280008         move.l 8(a0),d1
main+0x38           >202F001C         move.l 28(a7),d0
SrcDbg(DILIST): dilist @0xe69e6 : 5          display source line/disassembly  for 5 lines
main                >4E550000         link.w a5,#0    starting at an  absolute address
main+0x4            >48E7C8A0         movem.l d0-d1/d4/a0/a2,-(a7)
main+0x8            >2800             move.l d0,d4
  19: void main(argc,argv)
           ^main+0xa            >203CFFFFFFA4     move.l #-92,d0
SrcDbg(DILIST):
```

Disasm

Disassemble Memory

Syntax

```
di[sasm] [ [<C_expr>] [: [<count>]] ]
```

Usage

`Di[sasm]` disassembles memory starting at the address specified by <C_expr>.

If <count> is specified, then <count> machine lines will be disassembled and displayed. If <count> is not specified, then sixteen machine lines will be disassembled and displayed.

If Di[sasm] is entered without any arguments, the previous Di[sasm] command will be repeated. If there was no previous Di[sasm] command, an error message will be returned:

```
Error: no previous “disasm”
```

After a Di[sasm] command has been executed, SrcDbg displays the following prompt:

```
SrcDbg(DISASM):
```

To continue disassembly of the contents following the last address displayed by the previous Di[sasm] command, enter **[Return]**.

Examples

The following example shows how the Di[sasm] command is used to disassemble memory at the .pc register:

```
$ srcdbg where where.c /h0/usr/kathie                              fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: go main                                                    execute "where" until main
SrcDbg: disasm .pc                                                 disassemble 16 lines starting at
main+0xa       >203CFFFFFFA4   move.l #-92,d0                      the program counter register
main+0x10      >6100FEDA       bsr.w _stkchec
main+0x14      >4FEFFFF4       lea.l -12(a7),a7
main+0x18      >206F0010       movea.l 16(a7),a0
main+0x1c      >24680004       movea.l 4(a0),a2
main+0x20      >2004           move.l d4,d0
main+0x22      >61000040       bsr.w check_args
main+0x26      >4857           pea.l (a7)
main+0x28      >486F0008       pea.l 8(a7)
main+0x2c      >486F0010       pea.l 16(a7)
main+0x30      >206F001C       movea.l 28(a7),a0
main+0x34      >22280008       move.l 8(a0),d1
main+0x38      >202F001C       move.l 28(a7),d0
main+0x3c      >6100005C       bsr.w fork_dir
main+0x40      >4FEF000C       lea.l 12(a7),a7
main+0x44      >2F17           move.l (a7),-(a7)
SrcDbg(DISASM): disasm .pc + 0x3c : 4                              disassemble 4 lines starting
main+0x46      >2F2F0008       move.l 8(a7),-(a7)                  at the address ".pc + 0x3c"
main+0x4a      >222F0010       move.l 16(a7),d1
main+0x4e      >200A           move.l a2,d0
main+0x50      >610000FE       bsr.w read_dir_stuff
SrcDbg(DISASM): <return>                                           repeat previous DISASM command
main+0x54      >508F           addq.l #8,a7
main+0x56      >4FEF000C       lea.l 12(a7),a7
main+0x5a      >4CED0510FFF4   movem.l -12(a5),d4/a0/a2
main+0x60      >4E5D           unlk a5
SrcDbg(DISASM): disasm fork_dir : 3                                disassemble 3 lines starting
fork_dir       >4E550000       link.w a5,#0                        at the function "fork_dir"
fork_dir+0x4   >48E7CEB8       movem.l d0-d1/d4-d6/a0/a2-a4,-(a7)
fork_dir+0x8   >2440           movea.l d0,a2
SrcDbg(DISASM):
```

Dump

Displays Memory Contents

Syntax

```
d[ump] [ [<C_expr>] [: [<count>] [<format>] ]
```

Usage

D[ump] returns a formatted display of the physical contents starting at the address specified by <C_expr>.

If <count> is specified, then <count> lines of information will be displayed. If <count> is not specified, then sixteen lines of information will be displayed.

Three types of formatted display are available with the <format> option:

Format:	Data display:
f[loat]	floating point format
d[ouble]	double-length format
[e]x[tend]	extended floating point format

If no <format> is specified, then data is displayed 16 bytes per line in both hexadecimal and ASCII character format. Data that is non-displayable is represented by a period (.) in the ASCII area.

If no <C_expression> is specified, the last D[ump] command is repeated. If there was no previous D[ump] command, an error will be returned:

```
error:  no previous dump
```

After a D[ump] command has been executed, SrcDbg displays the following prompt:

```
SrcDbg(DUMP):
```

To continue "dumping" the contents following the last address displayed by the previous D[ump] command, enter **[Return]**.

The address contents displayed are absolute addresses in memory. Certain expressions may return an error because they try to access memory that does not belong to SrcDbg or the process being debugged.

Example

The following example shows how the D[ump] command is used:

```
SrcDbg: dump inbuf : 10                          dump 10 lines of memory starting at  inbuf
inbuf            - 2F68302F 434D4453 00000000 00000000  /h0/CMDS........
inbuf+0x10       - 00000000 00000000 00000000 00000000  ................
inbuf+0x20       - 00000000 00000000 00000000 00000000  ................
inbuf+0x30       - 00000000 00000000 00000000 00000000  ................
inbuf+0x40       - 00000000 00000000 00000000 00000000  ................
inbuf+0x50       - 00000000 00000000 00000000 00000000  ................
inbuf+0x60       - 00000000 00000000 00000000 00000000  ................
inbuf+0x70       - 00000000 00000000 00000000 00000000  ................
inbuf+0x80       - 00000000 00000000 00000000 00000000  ................
inbuf+0x90       - 00000000 00000000 00000000 00000000  ................
SrcDbg(DUMP): dump inbuf : 10 float              dump 10 lines of memory starting  at  inbuf
inbuf            - 2F68302F 2.111739533239287e-10          in floating point format
inbuf+0x10       - 00000000 0.
inbuf+0x20       - 00000000 0.
inbuf+0x30       - 00000000 0.
inbuf+0x40       - 00000000 0.
inbuf+0x50       - 00000000 0.
inbuf+0x60       - 00000000 0.
inbuf+0x70       - 00000000 0.
inbuf+0x80       - 00000000 0.
inbuf+0x90       - 00000000 0.
SrcDbg(DUMP): dump inbuf :extend 2               dump 2 lines of memory starting at inbuf
inbuf            - 2F68302F434D445300696E69 0.           in extended format
inbuf+0x30       - 000000000000000000000000 0.
SrcDbg(DUMP): <return>                           continue DUMP of 2 extend
inbuf+0x60       - 000000000000000000000000 0.
inbuf+0x90       - 000000000000000000000000 0.
SrcDbg(DUMP):
```

Gostop

Execute Machine Instruction in Current Subroutine

Syntax

```
gostop [<number>]
gs [<number>]
```

Usage

`Gostop` executes the specified number of machine instructions in the current subroutine. If <number> is not specified, SrcDbg executes one machine instruction.

After a Gostop instruction has been executed, SrcDbg displays the following prompt:

```
SrcDbg(GOSTOP):
```

To continue "gostopping" through the program, hit **[Return]** or re-enter the Gostop command. If <number> is specified, it remains in effect until a Gostop command with a different number (or no number) is entered.

The Gostop command has the same function as the T[race] command with one exception: if a "branch or jump to subroutine" is encountered while "gostopping" through a program, the subroutine is executed without tracing through it. To trace through subroutines, use the T[race] command.

Example

The following example shows how the Gostop command may be used:

```
$ srcdbg where where.c /h0/usr/kathie                                   fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: go main                                          execute "where" until main
File: "where.c"
Context: where\main
  19: void main(argc,argv)
           ^
SrcDbg: gostop                                       execute one  machine instruction
  19: void main(argc,argv)
           ^
dn: FFFFFFA4 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00030C80 00000000  00031958 0003194C   00031948 0003187C  00038000 00031868
pc:   1529F6  cc:  08 (-N---)
main+0x10           >6100FEDA         bsr.w _stkchec
SrcDbg(GOSTOP): <return>                               repeat last Gostop command
```

```
  19: void main(argc,argv)                              (execute next  machine instruction)
           ^
dn: FFFFFFA4 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00030C80 00000000  00031958 0003194C   00031948 0003187C  00038000 00031868
pc:   1529FA  cc:  00 (-----)
main+0x14           >4FEFFFF4         lea.l -12(a7),a7
SrcDbg(GOSTOP): gostop 3                                 execute 3 machine instructions
  23:   register char   *filename = argv[1];
                                          ^
dn: FFFFFFA4 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00030C80 00000000  00031958 0003194C   00031948 0003187C  00038000 0003185C
pc:   1529FE  cc:  00 (-----)
main+0x18           >206F0010         movea.l 16(a7),a0
  23:   register char   *filename = argv[1];
                                          ^
dn: FFFFFFA4 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00031954 00000000  00031958 0003194C   00031948 0003187C  00038000 0003185C
pc:   152A02  cc:  00 (-----)
main+0x1c           >24680004         movea.l 4(a0),a2
  26:   check_args(argc);
        ^
dn: FFFFFFA4 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00031954 00000000  00031888 0003194C   00031948 0003187C  00038000 0003185C
pc:   152A06  cc:  00 (-----)
main+0x20           >2004             move.l d4,d0
SrcDbg(GOSTOP): <return>                                   repeat last Gostop command
  26:   check_args(argc);                            (execute next 3 machine instructions)
        ^
dn: 00000003 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00031954 00000000  00031888 0003194C   00031948 0003187C  00038000 0003185C
pc:   152A08  cc:  00 (-----)
main+0x22           >61000040         bsr.w check_args
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
        ^
dn: 00000002 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00031954 00000000  00031888 0003194C   00031948 0003187C  00038000 0003185C
pc:   152A0C  cc:  09 (-N--C)
main+0x26           >4857             pea.l (a7)
  27:   fork_dir(argv,argv[2],&pid,&pipe,&stout);
        ^
dn: 00000002 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00031954 00000000  00031888 0003194C   00031948 0003187C  00038000 00031858
pc:   152A0E  cc:  09 (-N--C)
main+0x28           >486F0008         pea.l 8(a7)
SrcDbg(GOSTOP): option source                               turn off source line display
SrcDbg: gostop                                          execute next  machine instruction
dn: 00000002 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00031954 00000000  00031888 0003194C   00031948 0003187C  00038000 00031854
pc:   152A12  cc:  09 (-N--C)
main+0x2c           >486F0010         pea.l 16(a7)
SrcDbg(GOSTOP): <return>                                execute next  machine instruction
dn: 00000002 00031954  00000003 00000003   00000003 00000108  00001990 00000000
an: 00031954 00000000  00031888 0003194C   00031948 0003187C  00038000 00031850
pc:   152A16  cc:  09 (-N--C)
main+0x30           >206F001C         movea.l 28(a7),a0
```

Link

Link to Module

Syntax

```
li[nk] <module name>
```

Usage

`Li[nk]` links SrcDbg to <module_name> and places the module address in register .r7.

Examples

The following example shows how the Li[nk] command may be used:

```
SrcDbg: link t21            link SrcDbg to “t21”; place address of  t21 module in register .r7
SrcDbg: dump .r7                                          dump memory contents of register .r7
.r7              - 4AFC0001 00000078 0000001D 00000070  J|.....x.......p
.r7+0x10         - 05550F00 80000004 00000000 00000000  .U..............
.r7+0x20         - 00C80000 00000000 00000000 00003F8E  .H...........?.
.r7+0x30         - 001D2000 19011723 00640068 00280000  .. ....#.d.h.(..
.r7+0x40         - 00000000 0000001C 00000100 01010001  ................
.r7+0x50         - 1808180D 1B040117 03050807 000E0070  ...............p
.r7+0x60         - 11130904 53636600 68703236 36310000  ....Scf.hp2661..
.r7+0x70         - 74323100 001C10AB 4AFC0001 00000078  t21....+J|.....x
.r7+0x80         - 0000001D 00000070 05550F00 80000004  .......p.U......
.r7+0x90         - 00000000 00000000 00C80000 00000000  .........H......
.r7+0xa0         - 00000000 00003F8E 001D4000 19011823  ......?...@....#
.r7+0xb0         - 00640068 00280000 00000000 0000001C  .d.h.(..........
.r7+0xc0         - 00000100 01010001 1808180D 1B040117  ................
.r7+0xd0         - 03050807 000E0070 11130904 53636600  .......p....Scf.
.r7+0xe0         - 68703236 36310000 74323200 0065BF84  hp2661..t22..e?.
.r7+0xf0         - 4AFC0001 00000078 0000001D 00000070  J|.....x.......p
SrcDbg(DUMP): link where                                   link SrcDbg to program “where”
SrcDbg: dump .r7                                          dump memory contents of register .r7
btext            - 4AFC0001 0000387C 0000004C 00000048  J|....8|...L...H
btext+0x10       - 05550101 80010007 00000000 00000000  .U..............
btext+0x20       - 00000000 00000000 00000000 00000928  ...............(
btext+0x30       - 0000004E 0000019E 00000C80 00000C00  ...N............
btext+0x40       - 000037AC 0000385E 77686572 65002D46  ..7,..8^where.-F
_cstart+0x2      - 80102D46 80143D43 8018082B 00050014  ..-F..=C...+....
_cstart+0x12     - 670E2D4C 801A6608 2D790000 0000801A  g.-L..f.-y......
_cstart+0x22     - 4A85671E 08050000 661441F5 58004A68  J.g.....f.AuX.Jh
_cstart+0x32     - FFFE660A 598849E8 FFFC7001 60204235  .~f.Y.Ih.|p.` B5
_cstart+0x42     - 58FF204D D7EB000C 42A72F0B 74016100  X. MWk..B'/.t.a.
_cstart+0x52     - 21426076 43E80004 2D498BDA 74002260  !B`vCh..-I.Zt."`
_cstart+0x62     - 2E09670C D3CD4229 FFFF2089 528260EE  ..g.SMB).. .R.`n
_cstart+0x72     - 538067E0 4A826610 4A68FFFE 670A4228  S.g`J.f.Jh.~g.B(
_cstart+0x82     - FFFF2448 58886006 208D2448 52825282  ..$HX.`. .$HR.R.
_cstart+0x92     - 4A946718 28544A1C 66FCB5CC 631E0C1C  J.g.(TJ.f|5Lc...
_cstart+0xa2     - 00FC6618 528C2654 D7CD6014 0C2D00FC  .|f.R.&TWM`..-.|
SrcDbg(DUMP):
```

MFill

Fill Memory

Syntax

`mf[ill][n] <begin> : <end> : <value>`	byte fill
`mfw[n] <begin> : <end> : <value>`	word fill
`mfillword[n] <begin> : <end> : <value>`	word fill
`mfl[n] <begin> : <end> : <value>`	longword fill
`mfilllong[n] <begin> : <end> : <value>`	longword fill

Usage

`Mf[ill]` fills memory in the address range from <begin> to <end> with <value>. All arguments are <C_expr>'s.

`[n]` indicates that the fill is to be performed without respect to word/longword boundaries. That is, word and longword memory fills are done on a byte for byte basis. If the [n] option is not specified, word and longword memory fills must begin on even addresses on a non-68020 processor.

If the size of the fill specified from <begin> to <end> is not an even word or longword multiple (for a word or longword memory fill), the size is trimmed to the next lowest respective multiple.

If <value> starts with a " when using the byte fill size, all remaining characters are used as a fill string. The pattern is reused from the beginning if the fill count has not been exhausted.

Example

The following example show how the Mfill command may be used:

```
SrcDbg: mf inbuf : inbuf  + 0x16 : 0x11
```

fill bytes of memory from inbuf to inbuf+0x16 with 0x11

```
SrcDbg: dump inbuf : 5
```

dump 5 lines of memory starting at inbuf

```
inbuf             - 11111111 11111111 11111111 11111111  ................
inbuf+0x10        - 11111111 11111100 00000000 00000000  ................
inbuf+0x20        - 00000000 00000000 00000000 00000000  ................
inbuf+0x30        - 00000000 00000000 00000000 00000000  ................
inbuf+0x40        - 00000000 00000000 00000000 00000000  ................
SrcDbg(DUMP): mfw inbuf : inbuf + 0x16 :0x22
```

fill words of memory from inbuf to inbuf+0x16 with 0x22

```
SrcDbg: dump inbuf : 5
```

dump 5 lines of memory starting at inbuf

```
inbuf             - 00220022 00220022 00220022 00220022  ."."."."."."."."
inbuf+0x10        - 00220022 00221100 00000000 00000000  ."."."..........
inbuf+0x20        - 00000000 00000000 00000000 00000000  ................
inbuf+0x30        - 00000000 00000000 00000000 00000000  ................
inbuf+0x40        - 00000000 00000000 00000000 00000000  ................
SrcDbg(DUMP): mfl inbuf : inbuf + 0x16 : 0x33
```

fill longwords of memory from inbuf to inbuf+0x16 with 0x33

```
SrcDbg: dump inbuf : 5
```

dump 5 lines of memory starting at inbuf

```
inbuf             - 00000033 00000033 00000033 00000033  ...3...3...3...3
inbuf+0x10        - 00000033 00221100 00000000 00000000  ...3."..........
inbuf+0x20        - 00000000 00000000 00000000 00000000  ................
inbuf+0x30        - 00000000 00000000 00000000 00000000  ................
inbuf+0x40        - 00000000 00000000 00000000 00000000  ................
SrcDbg(DUMP): mfln inbuf + 1 : inbuf + 0x18 : 0x44
```

fill longwords of memory from (inbuf +1) to (inbuf+0x18) with 0x44 without respect to longword boundaries

```
SrcDbg: dump inbuf : 5
```

dump 5 lines of memory starting at inbuf

```
inbuf             - 00000000 44000000 44000000 44000000  ....D...D...D...
inbuf+0x10        - 44000000 44000000 44000000 00000000  D...D...D.......
inbuf+0x20        - 00000000 00000000 00000000 00000000  ................
inbuf+0x30        - 00000000 00000000 00000000 00000000  ................
inbuf+0x40        - 00000000 00000000 00000000 00000000  ................
SrcDbg(DUMP):
```

Msearch

Search Memory

Syntax

```
ms[earch][n] <begin> : <end> : <value> [: <mask>]              byte search
msw[n] <begin> : <end> : <value> [: <mask>]                    word search
msearchword[n] <begin> : <end> : <value> [: <mask>]            word search
msl[n] <begin> : <end> : <value> [: <mask>]                longword search
msearchlong[n] <begin> : <end> : <value> [: <mask>]        longword search
```

Usage

`Ms[earch]` searches memory from <begin> to <end> for <value> and displays each line where <value> is found. All arguments are <C_expr>'s.

`[n]` indicates that the search is to be performed without respect to word/longword boundaries. That is, word and longword memory searches are done on a byte for byte basis. If the [n] option is not specified, word and longword memory searches must begin on even addresses on a non-68020 processor.

If the range of the search specified from <begin> to <end> is not an even word or longword multiple (for a word or longword memory search), the range is trimmed to the next lowest respective multiple.

If <value> starts with a " (quotation mark) when using the byte search size, all remaining characters are used as a search string. The pattern is reused from the beginning if the search count has not been exhausted.

A <mask> may be specified to limit the comparison to only those bits set in the mask. If <mask> is not specified, the mask used is –1 (all bits set). The mask parameter is ignored for multiple character patterns.

Example

The following example shows how the Msearch command may be used:

```
SrcDbg: msw btext : etext : 1                 search for word-aligned “0001” in
                                              the memory area from btext to etext
btext+0x2            – 00010000 387C0000 004C0000 00480555  ....8|...L...H.U
sprintf+0x1c0        – 0001206F 000C58AF 000C2008 56802200  .. o..X/.. .V.“.
putc+0x40            – 000141EF 00072208 306A000E 2008206A  ..Ao..“.0j.. . j
fclose+0x20          – 0001670E 200A6154 28006006 4A6A000C  ..g. .aT(.‘.Jj..
fseek+0xb0           – 000167BE 7001B0AF 00206608 202A0008  ..g>p.0/. f. *..
ftell+0x2e           – 00017200 306A000E 20086100 0E50588F  ..r.0j.. .a..PX.
getw+0xb6            – 0001200A 7210D081 25400004 2200306A  .. .r.P.%@..“.0j
setbuf+0x68          – 00010012 202A0004 322A0012 48C1D081  .... *..2*..HAP.
_iobinit+0x16        – 00018370 3D7C0002 838A3D7C 0002838C  ...p=|....=|....
_T$LDiv+0x14         – 00016122 E20A6402 4480E20A 64024481  ..a“b.d.D.b.d.D.
getstat+0xc          – 0001672E 0C010006 673A0C01 00026710  ..g.....g:....g.
getstat+0x26         – 00016000 04564E40 008D6500 044E206F  ..‘..VN@..e..N o
lseek+0xc            – 00016716 0C010002 670672CB 6000028E  ..g.....g.rK‘...
modload+0xe          – 000164E4 4CDF0600 6000022A 48E76080  ..ddL_..‘..*Hg‘.
os9fork+0x24         – 00016704 262F0034 282F0030 08050000  ..g.&/.4(/.0....
_utinit+0x2          – 00014E75 4E752F05 7A004A80 6A047A08  ..NuNu/.z.J.j.z.
Tens16+0xa6          – 00013C67 0EF54646 D4973C9C D2B297D8  ..<g.uFFT.<.R2.X
_T$DInt+0x3a         – 00015247 E288E291 51CEFFF2 64125281  ..RGb.b.QN.rd.R.
_T$DMul+0x18e        – 000108C0 001F6030 0C828000 00006604  ...@..‘0......f.
PackD+0x60           – 00010101 01010101 01011111 01111101  ................
SrcDbg: msw btext : btext + 0x400 : 0x4e40    search for system calls (signified by the
                          word “4e40”) in the memory area from btext to btext + 0x400
_stkcheck+0x30       – 4E40008C 321F4E75 202E8000 90AE8008  N@..2.Nu .......
trapinit+0x1a        – 4E400021 64066100 254A6564 2F490014  N@.!d.a.%Jed/I..
trapinit+0xba        – 4E400006 4E400006 12D866FC 4E754E55  N@..N@...Xf|NuNU
trapinit+0xbe        – 4E400006 12D866FC 4E754E55 000048E7  N@...Xf|NuNU..Hg
SrcDbg: msl btext : etext : 0x4e400000 : 0xffffff80   search for  non–I/O system calls
                               (signified by the mask “0xffffff80”) in the memory area
                                                                  from btext to etext
trapinit+0x1a        – 4E400021 64066100 254A6564 2F490014  N@.!d.a.%Jed/I..
trapinit+0xba        – 4E400006 4E400006 12D866FC 4E754E55  N@..N@...Xf|NuNU
trapinit+0xbe        – 4E400006 12D866FC 4E754E55 000048E7  N@...Xf|NuNU..Hg
kill_dir+0x128       – 4E400029 245F6000 213448E7 C8302440  N@.)$_‘.!4HgH0$@
modlink+0xc          – 4E400000 64084CDF 06006000 024E200A  N@..d.L_..‘..N .
modload+0xc          – 4E400001 64E44CDF 06006000 022A48E7  N@..ddL_..‘..*Hg
munload+0x8          – 4E40001D 60000206 48E76080 22006600  N@..‘...Hg‘.“.f.
sbrk+0x14            – 4E400007 64000008 225F6000 01562D40  N@..d...“_‘..V-@
_srqmem+0x2          – 4E400028 650A2D40 8BCE200A 245F4E75  N@.(e.-@.N .$_Nu
_srtmem+0x8          – 4E400029 245F6000 00F848E7 60804E40  N@.)$_‘..xHg‘.N@
setpr+0x4            – 4E40000D 600000C2 48E76080 48E71C40  N@..‘..BHg‘.Hg.@
os9fork+0x3c         – 4E400005 4CDF0238 60000056 48E76080  N@..L_.8‘..VHg‘.
getpid+0x4           – 4E40000C 6000004A 48E76080 4E40000C  N@..‘..JHg‘.N@..
```

```
getuid+0x4          - 4E40000C 65000040 20016000 003848E7  N@..e..@ .`..8Hg
setuid+0x6          - 4E40001C 6000002A 48E76080 41FA0012  N@..`..*Hg`.Az..
_sigint+0x10        - 4E400009 60000012 2001206E 8BD24E90  N@..`... . n.RN.
_sigint+0x20        - 4E40001E 91C8C188 64062D41 800C70FF  N@...HA.d.-A..p.
SrcDbg: msln btext : etext : 0x4e400000 : 0xffffff80
```

same as last search, but without respect to longword boundaries

```
trapinit+0x1a       - 4E400021 64066100 254A6564 2F490014  N@.!d.a.%Jed/I..
trapinit+0xba       - 4E400006 4E400006 12D866FC 4E754E55  N@..N@...Xf|NuNU
trapinit+0xbe       - 4E400006 12D866FC 4E754E55 000048E7  N@...Xf|NuNU..Hg
kill_dir+0x106      - 4E400028 650C2041 2080200A 4CDF0500  N@.(e. A . .L_..
kill_dir+0x128      - 4E400029 245F6000 213448E7 C8302440  N@.)$_`.!4HgH0$@
modlink+0xc         - 4E400000 64084CDF 06006000 024E200A  N@..d.L_..`..N .
modload+0xc         - 4E400001 64E44CDF 06006000 022A48E7  N@..ddL_..`..*Hg
munlink+0x8         - 4E400002 245F6000 021648E7 60802040  N@..$_`...Hg`. @
munload+0x8         - 4E40001D 60000206 48E76080 22006600  N@..`...Hg`.".f.
ebrk+0x42           - 4E400028 204A245F 650001BA 2D488BC6  N@.( J$_e..:-H.F
sbrk+0x14           - 4E400007 64000008 225F6000 01562D40  N@..d..."_`..V-@
_srqmem+0x2         - 4E400028 650A2D40 8BCE200A 245F4E75  N@.(e.-@.N .$_Nu
_srtmem+0x8         - 4E400029 245F6000 00F848E7 60804E40  N@.)$_`..xHg`.N@
kill+0x4            - 4E400008 600000EC 48E76080 20407000  N@..`..lHg`. @p.
wait+0x8            - 4E400004 650000DE 24086700 00D64258  N@..e..^$.g..VBX
setpr+0x4           - 4E40000D 600000C2 48E76080 48E71C40  N@..`..BHg`.Hg.@
os9fork+0x36        - 4E400003 60044E40 00054CDF 02386000  N@..`.N@..L_.8`.
os9fork+0x3c        - 4E400005 4CDF0238 60000056 48E76080  N@..L_.8`..VHg`.
getpid+0x4          - 4E40000C 6000004A 48E76080 4E40000C  N@..`..JHg`.N@..
getuid+0x4          - 4E40000C 65000040 20016000 003848E7  N@..e..@ .`..8Hg
setuid+0x6          - 4E40001C 6000002A 48E76080 41FA0012  N@..`..*Hg`.Az..
_sigint+0x10        - 4E400009 60000012 2001206E 8BD24E90  N@..`... . n.RN.
_sigint+0x20        - 4E40001E 91C8C188 64062D41 800C70FF  N@...HA.d.-A..p.
_exit+0x2           - 4E400006 DEADDEAD 003C0001 4E754E75  N@..^-^-.<..NuNu
```

Symbol

Display C Expression as Symbolic Expression

Syntax

```
sy[mbol] [<C_expr>]
```

Usage

`Sy[mbol]` displays the result of the expression as a symbolic expression.

If no <C_expr> is specified, SrcDbg will repeat the last Sy[mbol] command.

Example

In the following example, where is forked with two parameters and executed until line 91. Two machine instructions are traced to move the address of filename into the register .d0. The Sy[mbol] command is then used to display the symbolic expression of registers .d0 and .d1.

```
$ srcdbg where where.c /h0/usr/kathie                                      fork "where"
Reading symbol file "where.dbg".
where.c
Reading symbol file "where.stb".
Context: where\_cstart
SrcDbg: go 91                                                execute "where" until line 91
File: "where.c"
Context: where\read_dir_stuff\$blk0\$blk1
  91:       if(!strcmp(p,filename)) puts(buf);
               ^
SrcDbg(STEP): t                                              trace one machine instruction
dn: 00036022 00037884  00000003 00000003   00000015 00000001  00000003 00036022
an: 000E6BE0 00000000  00037884 0003601E   00036021 00037848  0003E000 00037820
pc:    E6B9E  cc:  00 (-----)
read_dir_stuff+0x68 >2007            move.l d7,d0
SrcDbg(TRACE): <return>                                      trace one machine instruction
dn: 00036022 00037884  00000003 00000003   00000015 00000001  00000003 00036022
an: 000E6BE0 00000000  00037884 0003601E   00036021 00037848  0003E000 00037820
pc:    E6BA0  cc:  00 (-----)
read_dir_stuff+0x6a >61001532        bsr.w strcmp
SrcDbg(TRACE): sy .d0                     display symbolic expression of register ".d0"
where\inbuf+0x4
SrcDbg: sy .d1                            display symbolic expression of register ".d1"
where\_jmptbl+0xc04
SrcDbg:
```

Trace

Execute Machine Instruction

Syntax

```
t[race] [<number>]
```

Usage

T[race] executes the specified number of machine instructions. If <number> is not specified, T[race] executes a single instruction.

After a T[race] command has been executed, SrcDbg displays the following prompt:

```
SrcDbg(TRACE):
```

To continue “tracing” through a program, hit **[Return]** or re-enter the T[race] command. If <number> was previously specified, it remains in effect until a T[race] command with a different number (or no number) is executed.

If a “branch to subroutine” is encountered while tracing through a program, the subroutine is also traced through. To avoid tracing through functions, use the Gostop command.

If a breakpoint is encountered while tracing through a program, execution stops in the same manner as any execution command.

Example

The following example traces one machine instruction, turns on the source code display option and then traces two more instructions:

```
SrcDbg: trace                                            execute one machine instruction
dn: 00036022 00037884  00000003 00000003   00000015 00000001  00000003 00036022
an: 000E6BE0 00000000  00036022 0003601E   00036021 00037848  0003E000 0003780C
pc:    E80DA  cc:  00 (-----)
strcmp+0x6          >2641            movea.l d1,a3
SrcDbg(TRACE): option source             show source code with assembly level locations
SrcDbg: trace 2
  28:   read_dir_stuff(filename,pid,pipe,stout);
        ^
dn: 00036022 00037884  00000003 00000003   00000015 00000001  00000003 00036022
an: 000E6BE0 00000000  00036022 00037884   00036021 00037848  0003E000 0003780C
pc:    E80DC  cc:  00 (-----)
strcmp+0x8          >600A            bra.b strcmp+0x14
  28:   read_dir_stuff(filename,pid,pipe,stout);
        ^
dn: 00036022 00037884  00000003 00000003   00000015 00000001  00000003 00036022
an: 000E6BE0 00000000  00036022 00037884   00036021 00037848  0003E000 0003780C
pc:    E80E8  cc:  00 (-----)
strcmp+0x14         >1012            move.b (a2),d0
SrcDbg:
```

Chapter 11

SrcDbg Syntax and Commands

Syntax

```
srcdbg [-m=<mem>] <program> {<program arg>} ["]<[path redirection>["]
```

Commands

Asm

Asm displays the current register values and current machine instruction. A period (.) has the same affect as Asm.

A[ssign] <C_expr> = <C_expr>

A[ssign] sets the value of a program object.

B[reak] [<location_expr>] [:wh[en] <C_expr>] [:co[unt] <num>]

B[reak] sets a breakpoint at a specified line number, result of a <C_expr> or upon a when <C_expr> becoming true.

C[hange] [<C_expr>]

C[hange] changes bytes of memory starting at the result of <C_expr>.

Changelong [<C_expr>] Cl [<C_expr>]

Changelong changes longwords of memory starting at the result of <C_expr>.

Changeword [<C_expr>] Cw [<C_expr>]

Changeword changes words of memory starting at the result of <C_expr>.

C[h]c [<scope_expr>]

C[h]c changes the context for default name and location resolution.

C[h]d <pathlist>

C[h]d changes SrcDbg's current data directory. This command does not change the parent Shell's current directory.

C[h]x <pathlist>

C[h]x changes SrcDbg's current execution directory. This command does not change the parent Shell's current execution directory.

Con[text] [<scope_expr>]

Con[text] displays the complete scope expression of an object.

Dil[ist] [<location_expr>] [: [<count>]]

Dil[ist] displays the current C source line and the assembly code which maps to the current C source line starting at the address specified by <location_expr>.

Di[sasm] [<C_expr>] [: [<count>]]

Di[sasm] disassembles and displays memory starting at the address specified by <C_expr>. If <count> is specified, then <count> machine lines will be disassembled and displayed. If <count> is not specified, then sixteen machine lines will be disassembled and displayed.

D[ump] [<C_expr>] [: [<count>] [<format>]]]

D[ump] returns a formatted display of the physical contents starting at the address specified by <C_expr>. If <count> is specified, then <count> lines of information will be displayed. If <count> is not specified, then sixteen lines of information will be displayed.

Fi[nd] [<name>]

Fi[nd] displays all scope expressions found for <name>.

F[rame] [[±] <number>]

F[rame] displays stack frame information. If no arguments are specified, the name of each calling function, the location from which it was called and the frame number are displayed. If F[rame] is followed by <number>, the stack frame context will be changed to <number>.

Fo[rk] [<program> [<program arg>] ["]<[path redirection>["]]

Fo[rk] forks the specified program to begin a debugging session. If no Fo[rk] arguments are specified, Fo[rk] uses the last command arguments used in either the Fo[rk] command or the SrcDbg command line.

G[o] [<location_expr>] [:dis[play]]

G[o] begins program execution. If <location_expr> is specified, the program will run until <location_expr> is reached. If ":dis" is specified, each line of code executed is displayed.

Gostop [<number>]
Gs [<number>]

Gostop executes the specified number of machine instructions in the current subroutine.

H[elp]

H[elp] returns the help display.

I[nfo] [<scope>]

I[nfo] returns information about specified program objects and the current program location. If no argument is specified, I[nfo] returns the current location of the program.

K[ill] [<watch expr>] {[,<breakpoint>] [,<watch expr>]}
K[ill] [<breakpoint>] {[,<breakpoint>] [,<watch expr>]}

K[ill] removes all specified breakpoints and watch expressions. K[ill] uses SrcDbg notation for breakpoint and watch expressions (b1, b2, w1, w2, ...).

Li[nk] <module_name>

L[ink] links SrcDbg to <module_name> and places the module address in register .r7.

`L[ist] [<list arg> [,<list arg>]]`

`L[ist]` returns a source listing of the specified file or portion of the file. A list argument may be a file name, line number or a scope expression resulting in the following:

1. block number
2. function
3. line number

If a beginning line number is specified, a second line number may be used to specify the ending line number (i.e. list 1 10).

`Lo[cals]`

`Lo[cals]` displays the values of all local symbols.

`L[o]g <pathlist>`
`L[o]g : off`

`L[o]g` writes SrcDbg commands to <pathlist>. The specified pathlist is relative to the user's current data directory.
If ": off" is entered, the log file is closed.

`Mf[ill][n] <begin> : <end> : <value>`

`Mf[ill]` fills bytes of memory in the address range from <begin> to <end> with <value>. All arguments are <C_expr>'s. `[n]` indicates that the fill will not respect byte boundaries.

`Mfilllong[n] <begin> : <end> : <value>`
`Mfl[n] <begin> : end> : <value>`

`Mf[ill]` fills longwords of memory in the address range from <begin> to <end> with <value>. All arguments are <C_expr>'s. `[n]` indicates that the fill will not respect longword boundaries.

`Mfillword[n] <begin> : <end> : <value>`
`Mfw[n] <begin> : <end> : <value>`

`Mf[ill]` fills words of memory in the address range from <beg1n> to <end> with <value>. All arguments are <C_expr>'s. `[n]` indicates that the fill will not respect word boundaries.

`Ms[earch][n] <begin> : <end> : <value> [: <mask>]`

`Ms[earch]` searches bytes of memory from <begin> to <end> for <value> and displays each line where <value> is found. All arguments are <C_expr>'s. `[n]` indicates that the search will not respect byte boundaries.

`Msearchlong[n] <begin> : <end> : <value> [: <mask>]`
`Msl[n] <begin> : <end> : <value> [: <mask>]`

`Ms[earch]` searches longwords of memory from <begin> to <end> for <value> and displays each line where <value> is found. All arguments are <C_expr>'s. `[n]` indicates that the search will not respect longword boundaries.

`Msearchword[n] <begin> : <end> : <value> [: <mask>]`
`Msw[n] <begin> : <end> : <value> [: <mask>]`

`Ms[earch]` searches words of memory from <begin> to <end> for <value> and displays each line where <value> is found. All arguments are <C_expr>'s. `[n]` indicates that the search will not respect word boundaries.

Command	Description
N[ext] [<number>]	N[ext] executes the specified number of executable statements of the program. If no number is specified, one statement is executed. N[ext] executes functions as a single statement.
O[ption] {<options>}	O[ption] allows the user to set a variety of display and execution options.
P[rint] <C_expr>	P[rint] returns the value of the specified C expression.
Q[uit]	Q[uit] exits SrcDbg or aborts an interrupted function call.
Re[ad] [<pathlist>]	Re[ad] reads SrcDbg commands from <pathlist>. The <pathlist> is relative to the user's current data directory. The L[o]g command may be used to create the file referred to in <pathlist>.
R[eturn] [<number>]	R[eturn] executes the program until the current function returns to the calling function or a breakpoint is encountered. If a number is specified, R[eturn] executes until it returns to the specified number of callers above the current stack frame.
Se[tenv] <environment_name> <environment_definition>	Se[tenv] sets a shell-type environment variable for use by SrcDbg and SrcDbg's child processes. The arguments <environment_name> and <environment_definition> are strings that are stored in the environment list by the current shell.
S[tep] [<number>]	S[tep] executes the specified number of executable statements of the program. If no number is specified, one statement is executed.
Shell [shell command]	Shell forks a shell. If a Shell command is specified, the command is executed and control returns to SrcDbg.
Sy[mbol] [<C_expr>]	Sy[mbol] displays the result of the expression as a symbolic expression.
T[race] [<number>]	T[race] executes the specified number of machine instructions. If <number> is not specified, T[race] executes a single instruction.
Unse[tenv]	Unse[tenv] deletes an environment variable from the environment list.
W[atch] <C_expr>	W[atch] monitors the value of specified C expressions as the program is executed. Each watch expression is evaluated at each statement executed. The expression is displayed with its value each time the value changes.
?	Same as the Help command.
$	Same as the Shell command.

Appendix A

Error Message Descriptions

Error Message Overview

This appendix contains a list of C compiler and Preprocessor generated error messages, their probable causes and cross–references to the K & R Appendix A section number (in parentheses) to see for more specific information. There are four types of error messages. Each error message explanation includes the type of error message:

(W) Warning:
This is an informational message that indicates a potential deviation from accepted semantics or constraints. The output program may or may not provide expected results. An attempt to process the remainder of the program will be made.

(E) Error:
A violation of a semantics or syntax rule has occurred. It is not possible to create an executable program with such an error. An attempt to process the remainder of the program will be made.

(F) Fatal Error:
A message of this type indicates that an environment constraint has been exceeded. Such things as disk read or write errors, no more memory for the process, input buffer overflow, etc. are environment constraint errors. Processing is immediately halted as it is not possible to continue.

(C) Compiler Error:
The compiler has many built–in self checks to verify internal operation. This type of message usually indicates an error in the compiler. Sometimes this message is caused by an unusual program element that the compiler did not anticipate. An alternative method to express the program element may work around the problem. In any case, try to isolate the offending program section and submit it to your local Allen-Bradley sales representative or contact Allen-Bradley Software Support Services. Processing of the remainder of the program may or may not be possible, depending on the nature of the problem.

C Compiler Error Messages

The following table contains error messages that can be generated by the C compiler:

Compiler Error Messages:	Description
already a local variable (E):	Variable has already been declared at the current block level.
argument error (E):	Function argument declared as type function. Pointers to functions are allowed.
argument storage (E):	Function arguments may only be declared as storage class register.
bad character (E):	A character not in the C character set (probably a control character) was encountered in the source file.
both must be integral(E):	>> and << operands cannot be FLOAT or DOUBLE.
break error (E):	The break statement is allowed only inside a while, do, for or switch.
can't determine size (E):	The size of the object cast cannot be determined.
can't initialize unions (E):	Unions cannot appear as the object of an initializer.
can't open strings file (E):	A temporary file for constant string storage could not be opened. Likely cause is no permission in the directory or no room left on device.
can't take address (E):	& operator not allowed on a register variable. Operand must otherwise be an lvalue.
can't take size of bitfield (E):	The size of operator cannot be applied to bit fields.
cannot cast as function or array (E):	Type result of cast cannot be FUNCTION or ARRAY.
cannot evaluate size (E):	Could not determine size from declaration or initializer.
cannot initialize (E):	Storage class or type does not allow variable to be initialized.
cannot use void (E):	No operation can be performed on a void.
case value too large for type (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
compiler tag validation (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
compiler trouble (C):	Compiler detected something it couldn't handle. Try compiling the program again. If this error still occurs, contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
condition needed (E):	While, do, for, switch and if statements require a condition expression.
constant expression required (E):	Initializer expressions for static or external variables cannot reference variables. They may, however, refer to the address of a previously declared variable. This installation allows no initializer expressions unless all operands are of type INT or CHAR. Variables are not allowed for array dimensions or cases.
constant operator (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
constant overflow (E):	Input numeric constant was too large for the implied or explicit type.
continue error (E):	The continue statement is allowed only inside a while, do, or for block.

Compiler Error Messages:	Description
declaration mismatch (E):	This declaration conflicts with a previous one. This is typically caused by declaring a function to return a non–integer type after a reference has been made to the function. Depending on the declaration block's line structure, this error may be reported on the line following the erroneous declaration.
degenerative comparison with zero (W):	A comparison of the form u>=0 or u<0, where u is unsigned, is being done.
deref (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
deref storage (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
dimension mismatch (E):	An array has been declared twice, with conflicting bounds.
divide by zero (E):	Divide by zero occurred when evaluating a constant expression.
dumpstrings (F):	An error occurred during processing of the strings file. Likely cause is no more room on the output device.
duplicate cases (E):	All constant values used as a switch statement case must be unique.
duplicate member name (E):	A member identification name in a struct/union declaration has already appeared in this struct/union.
duplicate struct/union tags (E):	The tag name of this struct/union has already been defined at the current block level.
error writing assembly code file (E):	An error occurred when writing the output file. Likely cause is no room left on the output device.
? expected (E):	? is any character that was expected to appear here. Missing semicolons or braces cause this error.
expression missing (E):	An expression is required here.
expression too complex (C):	This expression could not be compiled by the compiler. If simplifying the expression (using temporaries) does not help, contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
expression with little effect (W):	This expression calculates a value that is never used.
function header missing (E):	Statement or expression encountered outside a function. Typically caused by mismatched braces.
function type error (E):	A function cannot be declared as returning an array, function, struct or union.
function unfinished (E):	End–of–file encountered before the end of function definition.
gen: unk opr (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
identifier missing (E):	Identifier name required here but none was found.
identifier name found in a cast (E):	Identifier name found in a cast. Only a cast types are allowed.
illegal declaration (E):	Declarations are allowed only at the beginning of a block.
illegal pointer/integer combination (W):	Mixing pointer and integer types may result in non–portable code.
illegal type combination (E):	The operators for the indicated operator do not have compatible types.
input line too long (F):	The source input line that was read is too long. The maximum length of a source input line is 512 characters.

Compiler Error Messages:	Description
invalid enumeration constant value (E):	Explicit values for enumerated type constants must be constant integral expressions.
Invalid bit field type (E):	Only int and unsigned are permissible types for bit fields.
invalid bit field size (E):	The width of a bit field must be a constant integral expression between 1 and 32.
label '<label>' undefined (E):	Goto label not defined in the current function.
label '<label>' unused (W):	The named label was defined but never referenced.
label required (E):	The goto statement requires a label identifier as an operand.
lvalue required (E):	Left side of assignment must be able to be ***stored into***. Array names, functions, etc. are not lvalues.
multiple defaults (E):	Only one default statement is allowed in a switch block.
multiple definition (E):	Identifier name was declared more than once in the same block level.
must be integral (E):	Type of object required here must be type int, char or pointer.
named twice (E):	Names in a function parameter list may appear only once.
no 'if' for 'else' (E):	Else statement found with no matching if. This is typically caused by extra or missing braces and/or semicolons.
no switch statement (E):	Case statements can only appear within a switch block.
not a function (E):	Primary in expression is not type "function returning...". If this is really a function call, the function name was declared differently elsewhere.
not a member of this struct/union (W):	The identifier given as the member name is not a member of the declared struct/union aggregate type.
not an argument (E):	Name does not appear in the function parameter list.
operand expected (E):	Unary operators require one operand, binary operators two. This is typically caused by misplaced parentheses, casts or operators.
operands have incompatible types (E):	The operands for the indicated operator do not have compatible types.
out of memory (F):	Compiler dynamic memory overflow. The compiler requires dynamic memory for symbol table entries, block level declarations and code generation. Three main factors affect this memory usage. Permanent declarations (those appearing on the outer block level (used in include files)) must be reserved from the dynamic memory for the duration of the compilation of the file. Each { causes the compiler to perform a block level recursion which may involve ***pushing down*** previous declarations which consume memory. Auto class initializers require saving expression trees until past the declarations which may be very memory–expensive if they exist. Avoiding excessive declarations, both permanent and inside compound statement blocks, conserves memory. If this error occurs on an auto initializer, try initializing the value in the code body.
pointer mismatch (E):	Pointers refer to different types. Use a cast if required.
pointer or integer required (E):	A pointer (of any type) or integer is required to the left of the –> operator.
pointer required (E):	Pointer operand required with unary * operator.
pointer type mismatch (W):	Mixing pointer types may not allow portability of code.
possible degenerate assignment in test (W):	This assignment in a conditional assignment in test expression may actually be a bug. Note the classic case: while(c = getchar() != EOF). It does not really behave as one might expect at first glance.
primary expected (E):	Primary expression required here.

Compiler Error Messages:	Description
reg free (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
rel op (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.
'return;' in non–void function (W):	A non–value returning return statement was found in a function returning a value of type other than void.
return value type mismatch (W):	The type of the expression returned by a return statement does not match the type of the declared function.
should be NULL (E):	Second and third expression of ?: conditional operator cannot be pointers to different types. If both are pointers, they must be of the same type or one of the two must be null.
****** STACK OVERFLOW **** (F):**	Compiler stack has overflowed. Most likely cause is very deep block–level nesting.
storage error (E):	Reg and auto storage classes may only be used within functions.
struct syntax, expecting brace (E):	Brace, comma, etc. is missing in a struct declaration.
struct or union inappropriate (E):	Struct or union cannot be used in this context.
struct/union size exceeds 32k (E):	The total size of a struct or union cannot exceed 32767 bytes.
struct/union member required (E):	The identifier on the right side of the arrow (–>) or period (.) operator must be a struct/union member identifier name.
struct/union object required (E):	The primary expression (left side) of the period (.) operator must be a struct/union object.
struct/union pointer mismatch (W):	Mixing struct/union pointer types may not allow portability of code.
struct/union pointer required (E):	The primary expression (left side) of the arrow (–>) operator must be a struct/union pointer.
struct/union type is not allowed (E):	A struct or union type is not allowed as an operand for the given operation.
syntax error (E):	Expression, declaration or statement is incorrectly formed.
syntax misplaced arg declaration list (E):	An argument declaration list is improperly placed in a function declarator.
third expression missing (E):	A question mark (?) must be followed by a colon (:) with expression. This error may be caused by unmatched parentheses or other errors in the expression.
too long (E):	Too many characters provided in a string initializing a character array.
too many braces (E):	Unmatched or unexpected braces encountered processing an initializer.
too many elements (E):	More data items supplied for aggregate level in initializer than members of the aggregate.
type expected (E):	A type name was expected, but not found.
typedef not a variable (E):	Typedef type name cannot be used in this manner.
undeclared identifier (E):	No declaration exists at any block level for this identifier.
undefined structure (E):	Union or struct declaration refers to an undefined structure name.
undefined struct/union tag referenced (E):	A struct or union object was referenced tag but has not yet been defined.A struct or union object was referenced tag but has not yet been defined.
uregfree (C):	Internal compiler error. If this message is reproducible, try to isolate the problem and and contact your local Allen-Bradley sales representative or Allen-Bradley Software Support Services.

Compiler Error Messages:	Description
unterminated character constant (E):	Unmatched ′ (character delimiters).
unterminated string (E):	Unmatched ″ (string delimiters).
while expected (E):	No while found for do statement.

Preprocessor Error Messages

The following table contains error messages that can be generated by the OS-9 Preprocessor:

Preprocessor Error Message:	Description
#if nesting too deep (E):	The maximum nesting for #if/#ifdef directives is 32 levels.
error writing output file (F):	An error occurred writing the output file. This is commonly caused by running out of space on the output storage device.
illegal #if macro name (E):	An illegal identifier was found in a #if/#ifdef directive.
illegal '#' (E):	An illegal directive was found on a pound sign (#) preprocessor line.
illegal macro name (E):	An illegal identifier was found during macro definition.
incorrect include file (E):	The file name given in an #include directive must be delimited by either double quotes (" ") or angle brackets (< >).
macro arguments required (E):	This macro was defined with arguments, but none were given when called.
macro definition error (E):	A syntax error was found during a macro definition. Macro dummy arguments must be a list of valid identifiers enclosed in parentheses. White space is required after the defining parenthesis.
missing #endif (E):	The end of the current file was reached and a pending #if/#ifdef or #else was in effect.
no #if for #else (E):	An #else directive was encountered without a corresponding #if/#ifdef.
out of memory (E):	No more memory is available to continue processing.
redefined macro (W):	The indicated macro name has already been defined. Use #undef <name> (carefully) if it is intended to redefine a macro name.
source file read error (F):	An error occurred reading the input source file.
source file too long (F):	The maximum length of an input line is 512 characters.
too few macro arguments (E):	A macro was called without enough arguments to match the macro definition.
too many #endifs (E):	An #endif directive was encountered before a corresponding #if/#ifdef.
too many macro arguments (E):	A macro was called with more arguments than given in the macro definition.

ALLEN-BRADLEY
A ROCKWELL INTERNATIONAL COMPANY

As a subsidiary of Rockwell International, one of the world's largest technology companies — Allen-Bradley meets today's challenges of industrial automation with over 85 years of practical plant-floor experience. More than 11,000 employees throughout the world design, manufacture and apply a wide range of control and automation products and supporting services to help our customers continuously improve quality, productivity and time to market. These products and services not only control individual machines but integrate the manufacturing process, while providing access to vital plant floor data that can be used to support decision-making throughout the enterprise.

With offices in major cities worldwide

WORLD HEADQUARTERS
Allen-Bradley
1201 South Second Street
Milwaukee, WI 53204 USA
Tel: (1) 414 382-2000
Telex: 43 11 016
FAX: (1) 414 382-4444

EUROPE/MIDDLE EAST/AFRICA HEADQUARTERS
Allen-Bradley Europe B.V.
Amsterdamseweg 15
1422 AC Uithoorn
The Netherlands
Tel: (31) 2975/43500
Telex: (844) 18042
FAX: (31) 2975/60222

ASIA/PACIFIC HEADQUARTERS
Allen-Bradley (Hong Kong) Limited
Room 1006, Block B,
Sea View Estate
28 Watson Road
Hong Kong
Tel: (852) 887-4788
Telex: (780) 64347
FAX: (852) 510-9436

CANADA HEADQUARTERS
Allen-Bradley Canada Limited
135 Dundas Street
Cambridge, Ontario N1R 5X1
Canada
Tel: (1) 519 623-1810
FAX: (1) 519 623-8930

LATIN AMERICA HEADQUARTERS
Allen-Bradley
1201 South Second Street
Milwaukee, WI 53204 USA
Tel: (1) 414 382-2000
Telex: 43 11 016
FAX: (1) 414 382-2400

Publication 1771-6.5.104 May 1993
Supersedes 1771-6.5.104 September 1992

PN 955114-96
Copyright 1993 Allen-Bradley Company, Inc. Printed in USA