SYSTEM V APPLICATION BINARY INTERFACE

Intel386™ Architecture Processor Supplement

Fourth Edition

Copyright © 1990–1996 The Santa Cruz Operation, Inc. All rights reserved.

Copyright © 1990–1992 AT&T. All rights reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, California, 95060, USA. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

SCO® UnixWare® is commercial computer software and, together with any related documentation, is subject to the restrictions on US Government use as set forth below. If this procurement is for a DOD agency, the following DFAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

If this procurement is for a civilian government agency, this FAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: This computer software is submitted with restricted rights under Government Contract No. _____ (and Subcontract No. _____, if appropriate). It may not be used, reproduced, or disclosed by the Government except as provided in paragraph (g)(3)(i) of FAR Clause 52.227-14 alt III or as otherwise expressly stated in the contract. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

If any copyrighted software accompanies this publication, it is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software.

ACKNOWLEDGEMENTS

"We acknowledge the contributions of the 88OPEN Consortium Ltd., portions of whose *System V ABI Implementation Guide for the M88000 Processor* and the *System V ABI M88000 Processor Networking Supplement* have been incorporated in this section of the ABI with permission."

TRADEMARKS

SCO, the SCO logo, The Santa Cruz Operation, and UnixWare are trademarks or registered trademarks of The Santa Cruz Operation, Inc. in the USA and other countries. Intel386, Intel486, and Pentium are trademarks of Intel Corporation. UNIX is a registered trademark in the USA and other countries, licensed exclusively through X/Open Company Limited. Motif is a trademark of the Open Software Foundation, Inc. NeWS is a registered trademark of Sun Microsystems, Inc. X11 and X Window System are trademarks of Massachusetts Institute of Technology. All other brand and product names are or may be trademarks of, and are used to identify products or services of, their respective owners.

Contents

Table of Contents

Table of Contents
INTRODUCTION
SOFTWARE INSTALLATION
LOW-LEVEL SYSTEM INFORMATION
OBJECT FILES
PROGRAM LOADING AND DYNAMIC LINKING
LIBRARIES
DEVELOPMENT ENVIRONMENT
EXECUTION ENVIRONMENT
Index

1 INTRODUCTION
The Intel386 Architecture and the System V ABI 1-1
How to Use the Intel386 Architecture ABI Supplement 1-2

2 SOFTWARE INSTALLATION
Software Distribution Formats 2-1

3 LOW-LEVEL SYSTEM INFORMATION
Machine Interface 3-1
Function Calling Sequence 3-9
Operating System Interface 3-20
Coding Examples 3-34

4 OBJECT FILES

ELF Header 4-1
Sections 4-2
Symbol Table 4-3
Relocation 4-4

5 PROGRAM LOADING AND DYNAMIC LINKING

Program Loading 5-1
Dynamic Linking 5-5

6 LIBRARIES

Shared Library Names 6-1
C Library 6-2
System Data Interfaces 6-5

7 DEVELOPMENT ENVIRONMENT

Development Commands 7-1
Software Packaging Tools 7-2

8 EXECUTION ENVIRONMENT

Application Environment 8-1

IN Index

Index IN-1

Figures and Tables

Figure 3-1: Scalar Types 3-2
Figure 3-2: Structure Smaller Than a Word 3-4
Figure 3-3: No Padding 3-4
Figure 3-4: Internal Padding 3-4
Figure 3-5: Internal and Tail Padding 3-5
Figure 3-6: `union` Allocation 3-5
Figure 3-7: Bit-Field Ranges 3-6
Figure 3-8: Bit Numbering 3-7
Figure 3-9: Right-to-Left Allocation 3-7
Figure 3-10: Boundary Alignment 3-7
Figure 3-11: Storage Unit Sharing 3-8
Figure 3-12: `union` Allocation 3-8
Figure 3-13: Unnamed Bit-Fields 3-8
Figure 3-14: Processor Registers 3-9
Figure 3-15: Standard Stack Frame 3-10
Figure 3-16: Function Prologue 3-13
Figure 3-17: Function Epilogue 3-14
Figure 3-18: Stack Contents for Functions Returning `struct/union` 3-15
Figure 3-19: Function Prologue (Returning `struct/union`) 3-16
Figure 3-20: Function Epilogue 3-16
Figure 3-21: Integral and Pointer Arguments 3-17
Figure 3-22: Floating-Point Arguments 3-18
Figure 3-23: Structure and Union Arguments 3-18
Figure 3-24: Virtual Address Configuration 3-20
Figure 3-25: Conventional Segment Arrangements 3-22
Figure 3-26: `_exit` System Trap 3-24
Figure 3-27: Hardware Exceptions and Signals 3-25
Figure 3-28: Declaration for `main` 3-26
Figure 3-29: EFLAGS Register Fields 3-27
Figure 3-30: Floating-Point Control Word 3-27
Figure 3-31: Initial Process Stack 3-28
Figure 3-32: Auxiliary Vector 3-30
Figure 3-33: Auxiliary Vector Types, `a_type` 3-30
Figure 3-34: AT_FPHW values 3-32
Figure 3-35: Example Process Stack 3-33

Figure 3-36: Calculating Global Offset Table Address 3-36
Figure 3-37: Position-Independent Function Prologue 3-37
Figure 3-38: Absolute Data Access 3-37
Figure 3-39: Position-Independent Data Access 3-38
Figure 3-40: Position-Independent Static Data Access 3-39
Figure 3-41: Absolute Direct Function Call 3-39
Figure 3-42: Position-Independent Direct Function Call 3-40
Figure 3-43: Absolute Indirect Function Call 3-40
Figure 3-44: Position-Independent Indirect Function Call 3-40
Figure 3-45: Branch Instruction, All Models 3-41
Figure 3-46: Absolute `switch` Code 3-41
Figure 3-47: Position-Independent `switch` Code 3-42
Figure 3-48: C Stack Frame 3-43
Figure 3-49: Dynamic Stack Allocation 3-45
Figure 4-1: Intel386 Identification, `e_ident` 4-1
Figure 4-2: Special Sections 4-2
Figure 4-3: Relocatable Fields 4-4
Figure 4-4: Relocation Types 4-5
Figure 5-1: Executable File 5-1
Figure 5-2: Program Header Segments 5-2
Figure 5-3: Process Image Segments 5-3
Figure 5-4: Example Shared Object Segment Addresses 5-4
Figure 5-5: Global Offset Table 5-6
Figure 5-6: Absolute Procedure Linkage Table 5-8
Figure 5-7: Position-Independent Procedure Linkage Table 5-8
Figure 6-1: Shared Library Names 6-1
Figure 6-2: `libc` Additional Required Entry Points 6-2
Figure 6-3: `libc`, Support Routines 6-3
Figure 6-4: `libc`, Global External Data Symbols 6-4
Figure 6-5: `<aio.h>*` 6-6
Figure 6-6: `<assert.h>` 6-6
Figure 6-7: `<ctype.h>` 6-7
Figure 6-8: `<dirent.h>` 6-8
Figure 6-9: `<dlfcn.h>*` 6-8
Figure 6-10: `<elf.h>*, Part 1 of 6` 6-9
Figure 6-11: `<elf.h>*, Part 2 of 6` 6-10
Figure 6-12: `<elf.h>*, Part 3 of 6` 6-11
Figure 6-13: `<elf.h>*, Part 4 of 6` 6-12
Figure 6-14: `<elf.h>*, Part 5 of 6` 6-13
Figure 6-15: `<elf.h>*, Part 6 of 6` 6-14
Figure 6-16: `<errno.h>`, Part 1 of 3 6-15
Figure 6-17: `<errno.h>`, Part 2 of 3 6-16
Figure 6-18: `<errno.h>`, Part 3 of 3 6-17
Figure 6-19: `<fcntl.h>`, Part 1 of 2 6-18
Figure 6-20: `<fcntl.h>`, Part 2 of 2 6-19

Figure 6-21: <float.h>, Single-Precision 6-20
Figure 6-22: <float.h>, Double-Precision 6-20
Figure 6-23: <float.h>, Extended-Precision 6-21
Figure 6-24: <fmtmsg.h>, Part 1 of 2 6-21
Figure 6-25: <fmtmsg.h>, Part 2 of 2 6-22
Figure 6-26: <fnmatch.h>* 6-22
Figure 6-27: <ftw.h> 6-23
Figure 6-28: <glob.h>* 6-24
Figure 6-29: <grp.h> 6-25
Figure 6-30: <iconv.h>* 6-25
Figure 6-31: <sys/ipc.h> 6-26
Figure 6-32: <langinfo.h>, Part 1 of 2 6-27
Figure 6-33: <langinfo.h>, Part 2 of 2 6-28
Figure 6-34: <limits.h>, Part 1 of 2 6-29
Figure 6-35: <limits.h>, Part 2 of 2 6-30
Figure 6-36: <locale.h> 6-31
Figure 6-37: <lwpsynch.h>* 6-32
Figure 6-38: <machlock.h>* 6-32
Figure 6-39: <math.h> 6-32
Figure 6-40: <sys/mman.h> 6-33
Figure 6-41: <sys/mod.h>* 6-34
Figure 6-42: <sys/mount.h> 6-35
Figure 6-43: <sys/msg.h> 6-36
Figure 6-44: <netconfig.h>, Part 1 of 2 6-37
Figure 6-45: <netconfig.h>, Part 2 of 2 6-38
Figure 6-46: <netdir.h>, Part 1 of 2 6-39
Figure 6-47: <netdir.h>, Part 2 of 2 6-40
Figure 6-48: <nl_types.h> 6-40
Figure 6-49: <sys/param.h> 6-41
Figure 6-50: <poll.h> 6-42
Figure 6-51: <sys/priocntl.h>* 6-43
Figure 6-52: <sys/procset.h> 6-44
Figure 6-53: <pwd.h> 6-45
Figure 6-54: <regex.h>*, Part 1 of 2 6-46
Figure 6-55: <regex.h>*, Part 2 of 2 6-47
Figure 6-56: <sys/resource.h> 6-48
Figure 6-57: <rpc.h>, Part 1 of 16 6-49
Figure 6-58: <rpc.h>, Part 2 of 16 6-50
Figure 6-59: <rpc.h>, Part 3 of 16 6-51
Figure 6-60: <rpc.h>, Part 4 of 16 6-52
Figure 6-61: <rpc.h>, Part 5 of 16 6-53
Figure 6-62: <rpc.h>, Part 6 of 16 6-54
Figure 6-63: <rpc.h>, Part 7 of 16 6-55
Figure 6-64: <rpc.h>, Part 8 of 16 6-56
Figure 6-65: <rpc.h>, Part 9 of 16 6-57

Figure 6-66: <rpc.h>, Part 10 of 16 6-58
Figure 6-67: <rpc.h>, Part 11 of 16 6-59
Figure 6-68: <rpc.h>, Part 12 of 16 6-60
Figure 6-69: <rpc.h>, Part 13 of 16 6-60
Figure 6-70: <rpc.h>, Part 14 of 16 6-61
Figure 6-71: <rpc.h>, Part 15 of 16 6-62
Figure 6-72: <rpc.h>, Part 16 of 16 6-63
Figure 6-73: <rtpriocntl.h>* 6-63
Figure 6-74: <search.h> 6-64
Figure 6-75: <sys/sem.h> 6-65
Figure 6-76: <setjmp.h> 6-66
Figure 6-77: <sys/shm.h> 6-66
Figure 6-78: <signal.h>, Part 1 of 3 6-67
Figure 6-79: <signal.h>, Part 2 of 3 6-68
Figure 6-80: <signal.h>, Part 3 of 3 6-69
Figure 6-81: <sys/siginfo.h>, Part 1 of 5 6-69
Figure 6-82: <sys/siginfo.h>, Part 2 of 5 6-70
Figure 6-83: <sys/siginfo.h>, Part 3 of 5 6-71
Figure 6-84: <sys/siginfo.h>, Part 4 of 5 6-72
Figure 6-85: <sys/siginfo.h>*, Part 5 of 5 6-72
Figure 6-86: <sys/stat.h>, Part 1 of 2 6-73
Figure 6-87: <sys/stat.h>, Part 2 of 2 6-74
Figure 6-88: <sys/statvfs.h> 6-75
Figure 6-89: <stdarg.h> 6-76
Figure 6-90: <stddef.h> 6-76
Figure 6-91: <stdio.h>, Part 1 of 2 6-77
Figure 6-92: <stdio.h>, Part 2 of 2 6-78
Figure 6-93: <stdlib.h> 6-79
Figure 6-94: <stropts.h>, Part 1 of 6 6-80
Figure 6-95: <stropts.h>, Part 2 of 6 6-81
Figure 6-96: <stropts.h>, Part 3 of 6 6-82
Figure 6-97: <stropts.h>, Part 4 of 6 6-83
Figure 6-98: <stropts.h>, Part 5 of 6 6-84
Figure 6-99: <stropts.h>, Part 6 of 6 6-85
Figure 6-100: <synch.h>*, Part 1 of 3 6-86
Figure 6-101: <synch.h>*, Part 2 of 3 6-87
Figure 6-102: <synch.h>*, Part 3 of 3 6-88
Figure 6-103: <sys/sysi86.h> 6-88
Figure 6-104: <termios.h>, Part 1 of 10 6-89
Figure 6-105: <termios.h>, Part 2 of 10 6-90
Figure 6-106: <termios.h>, Part 3 of 10 6-91
Figure 6-107: <termios.h>, Part 4 of 10 6-92
Figure 6-108: <termios.h>, Part 5 of 10 6-93
Figure 6-109: <termios.h>, Part 6 of 10 6-94
Figure 6-110: <termios.h>, Part 7 of 10 6-95

Figure 6-111: <termios.h>, Part 8 of 10 6-96
Figure 6-112: <termios.h>, Part 9 of 10 6-97
Figure 6-113: <termios.h>, Part 10 of 10 6-98
Figure 6-114: <thread.h>*, Part 1 of 2 6-99
Figure 6-115: <thread.h>*, Part 2 of 2 6-100
Figure 6-116: <sys/ticlts.h> 6-100
Figure 6-117: <sys/ticots.h> 6-101
Figure 6-118: <sys/ticotsord.h> 6-101
Figure 6-119: <time.h>* 6-102
Figure 6-120: <sys/time.h> 6-103
Figure 6-121: <sys/times.h> 6-104
Figure 6-122: <tiuser.h>, Error Return Values 6-105
Figure 6-123: <tiuser.h>, Event Bitmasks 6-106
Figure 6-124: <tiuser.h>, Flags 6-106
Figure 6-125: <tiuser.h>, Service Types 6-107
Figure 6-126: <tiuser.h>, Values for flags field in t_info structure 6-107
Figure 6-127: <tiuser.h>, Transport Interface Data Structures, 1 of 2 6-108
Figure 6-128: <tiuser.h>, Transport Interface Data Structures, 2 of 2 6-109
Figure 6-129: <tiuser.h>, Structure Types 6-110
Figure 6-130: <tiuser.h>, Fields of Structures 6-110
Figure 6-131: <tiuser.h>, Transport Interface States 6-111
Figure 6-132: <tiuser.h>, User-level Events 6-112
Figure 6-133: <tspriocntl.h>* 6-113
Figure 6-134: <sys/types.h> 6-114
Figure 6-135: <ucontext.h>, Part 1 of 2 6-115
Figure 6-136: <ucontext.h>, Part 2 of 2 6-116
Figure 6-137: <sys/uio.h> 6-117
Figure 6-138: <ulimit.h> 6-117
Figure 6-139: <unistd.h>, Part 1 of 2 6-117
Figure 6-140: <unistd.h>, Part 2 of 2 6-119
Figure 6-141: <utime.h> 6-119
Figure 6-142: <sys/utsname.h> 6-120
Figure 6-143: <wait.h> 6-121
Figure 6-144: <wchar.h> 6-122
Figure 6-145: <wctype.h>*, Part 1 of 3 6-123
Figure 6-146: <wctype.h>*, Part 2 of 3 6-124
Figure 6-147: <wctype.h>*, Part 3 of 3 6-125
Figure 6-148: <wordexp.h>* 6-126
Figure 6-149: <X11/Atom.h>, Part 1 of 3 6-128
Figure 6-150: <X11/Atom.h>, Part 2 of 3 6-129
Figure 6-151: <X11/Atom.h>, Part 3 of 3 6-130
Figure 6-152: <X11/Composite.h> 6-131
Figure 6-153: <X11/Constraint.h> 6-131
Figure 6-154: <X11/Core.h> 6-131
Figure 6-155: <X11/cursorfont.h>, Part 1 of 3 6-132

Figure 6-156: <X11/cursorfont.h>, Part 2 of 3 6-133
Figure 6-157: <X11/cursorfont.h>, Part 3 of 3 6-134
Figure 6-158: <X11/Intrinsic.h>, Part 1 of 6 6-135
Figure 6-159: <X11/Intrinsic.h>, Part 2 of 6 6-136
Figure 6-160: <X11/Intrinsic.h>, Part 3 of 6 6-137
Figure 6-161: <X11/Intrinsic.h>, Part 4 of 6 6-138
Figure 6-162: <X11/Intrinsic.h>, Part 5 of 6 6-139
Figure 6-163: <X11/Intrinsic.h>, Part 6 of 6 6-140
Figure 6-164: <X11/Object.h> 6-140
Figure 6-165: <X11/RectObj.h> 6-140
Figure 6-166: <X11/extensions/shape.h>* 6-141
Figure 6-167: <X11/Shell.h> 6-141
Figure 6-168: <X11/Vendor.h> 6-141
Figure 6-169: <X11/X.h>, Part 1 of 12 6-142
Figure 6-170: <X11/X.h>, Part 2 of 12 6-143
Figure 6-171: <X11/X.h>, Part 3 of 12 6-144
Figure 6-172: <X11/X.h>, Part 4 of 12 6-145
Figure 6-173: <X11/X.h>, Part 5 of 12 6-146
Figure 6-174: <X11/X.h>, Part 6 of 12 6-147
Figure 6-175: <X11/X.h>, Part 7 of 12 6-148
Figure 6-176: <X11/X.h>, Part 8 of 12 6-149
Figure 6-177: <X11/X.h>, Part 9 of 12 6-150
Figure 6-178: <X11/X.h>, Part 10 of 12 6-151
Figure 6-179: <X11/X.h>, Part 11 of 12 6-152
Figure 6-180: <X11/X.h>, Part 12 of 12 6-153
Figure 6-181: <X11/Xcms.h>, Part 1 of 5 6-154
Figure 6-182: <X11/Xcms.h>, Part 2 of 5 6-155
Figure 6-183: <X11/Xcms.h>, Part 3 of 5 6-156
Figure 6-184: <X11/Xcms.h>, Part 4 of 5 6-157
Figure 6-185: <X11/Xcms.h>, Part 5 of 5 6-158
Figure 6-186: <X11/Xlib.h>, Part 1 of 27 6-159
Figure 6-187: <X11/Xlib.h>, Part 2 of 27 6-159
Figure 6-188: <X11/Xlib.h>, Part 3 of 27 6-160
Figure 6-189: <X11/Xlib.h>, Part 4 of 27 6-161
Figure 6-190: <X11/Xlib.h>, Part 5 of 27 6-162
Figure 6-191: <X11/Xlib.h>, Part 6 of 27 6-163
Figure 6-192: <X11/Xlib.h>, Part 7 of 27 6-164
Figure 6-193: <X11/Xlib.h>, Part 8 of 27 6-165
Figure 6-194: <X11/Xlib.h>, Part 9 of 27 6-166
Figure 6-195: <X11/Xlib.h>, Part 10 of 27 6-167
Figure 6-196: <X11/Xlib.h>, Part 11 of 27 6-168
Figure 6-197: <X11/Xlib.h>, Part 12 of 27 6-169
Figure 6-198: <X11/Xlib.h>, Part 13 of 27 6-170
Figure 6-199: <X11/Xlib.h>, Part 14 of 27 6-171
Figure 6-200: <X11/Xlib.h>, Part 15 of 27 6-172

Figure 6-201: <X11/Xlib.h>, Part 16 of 27 6-173
Figure 6-202: <X11/Xlib.h>, Part 17 of 27 6-174
Figure 6-203: <X11/Xlib.h>, Part 18 of 27 6-175
Figure 6-204: <X11/Xlib.h>, Part 19 of 27 6-176
Figure 6-205: <X11/Xlib.h>, Part 20 of 27 6-177
Figure 6-206: <X11/Xlib.h>, Part 21 of 27 6-178
Figure 6-207: <X11/Xlib.h>, Part 22 of 27 6-179
Figure 6-208: <X11/Xlib.h>, Part 23 of 27 6-180
Figure 6-209: <X11/Xlib.h>, Part 24 of 27 6-181
Figure 6-210: <X11/Xlib.h>, Part 25 of 27 6-182
Figure 6-211: <X11/Xlib.h>, Part 26 of 27 6-183
Figure 6-212: <X11/Xlib.h>, Part 27 of 27 6-184
Figure 6-213: <X11/Xresource.h>, Part 1 of 2 6-185
Figure 6-214: <X11/Xresource.h>, Part 2 of 2 6-186
Figure 6-215: <X11/Xutil.h>, Part 1 of 5 6-187
Figure 6-216: <X11/Xutil.h>, Part 2 of 5 6-188
Figure 6-217: <X11/Xutil.h>, Part 3 of 5 6-189
Figure 6-218: <X11/Xutil.h>, Part 4 of 5 6-190
Figure 6-219: <X11/Xutil.h>, Part 5 of 5 6-191
Figure 6-220: <Xm/ArrowB.h>* 6-193
Figure 6-221: <Xm/ArrowBG.h>* 6-193
Figure 6-222: <Xm/BulletinB.h>* 6-193
Figure 6-223: <Xm/CascadeB.h>* 6-193
Figure 6-224: <Xm/CascadeBG.h>* 6-194
Figure 6-225: <Xm/Command.h>* 6-194
Figure 6-226: <Xm/CutPaste.h>* 6-195
Figure 6-227: <Xm/DialogS.h>* 6-195
Figure 6-228: <Xm/Display.h>* 6-196
Figure 6-229: <Xm/DragC.h>*, Part 1 of 4 6-197
Figure 6-230: <Xm/DragC.h>*, Part 2 of 4 6-198
Figure 6-231: <Xm/DragC.h>*, Part 3 of 4 6-199
Figure 6-232: <Xm/DragC.h>*, Part 4 of 4 6-200
Figure 6-233: <Xm/DragIcon.h>* 6-201
Figure 6-234: <Xm/DragOverS.h>* 6-201
Figure 6-235: <Xm/DrawingA.h>* 6-202
Figure 6-236: <Xm/DrawnB.h>* 6-202
Figure 6-237: <Xm/DropSMgr.h>*, Part 1 of 2 6-203
Figure 6-238: <Xm/DropSMgr.h>*, Part 2 of 2 6-204
Figure 6-239: <Xm/DropTrans.h>* 6-205
Figure 6-240: <Xm/FileSB.h>* 6-205
Figure 6-241: <Xm/Form.h>* 6-205
Figure 6-242: <Xm/Frame.h>* 6-206
Figure 6-243: <Xm/Label.h>* 6-206
Figure 6-244: <Xm/LabelG.h>* 6-206
Figure 6-245: <Xm/List.h>* 6-207

Figure 6-246: <Xm/MainW.h>* 6-207
Figure 6-247: <Xm/MenuShell.h>* 6-207
Figure 6-248: <Xm/MessageB.h>* 6-208
Figure 6-249: <Mrm/MrmPublic.h>*, Part 1 of 3 6-209
Figure 6-250: <Mrm/MrmPublic.h>*, Part 2 of 3 6-210
Figure 6-251: <Mrm/MrmPublic.h>*, Part 3 of 3 6-211
Figure 6-252: <Xm/MwmUtil.h>*, Part 1 of 3 6-212
Figure 6-253: <Xm/MwmUtil.h>*, Part 2 of 3 6-213
Figure 6-254: <Xm/MwmUtil.h>*, Part 3 of 3 6-214
Figure 6-255: <Xm/PanedW.h>* 6-214
Figure 6-256: <Xm/PushB.h>* 6-215
Figure 6-257: <Xm/PushBG.h>* 6-215
Figure 6-258: <Xm/RepType.h>* 6-215
Figure 6-259: <Xm/RowColumn.h>* 6-216
Figure 6-260: <Xm/Scale.h>* 6-216
Figure 6-261: <Xm/Screen.h>* 6-216
Figure 6-262: <Xm/ScrollBar.h>* 6-216
Figure 6-263: <Xm/ScrolledW.h>* 6-217
Figure 6-264: <Xm/SelectioB.h>* 6-217
Figure 6-265: <Xm/SeparatoG.h>* 6-217
Figure 6-266: <Xm/Separator.h>* 6-217
Figure 6-267: <Xm/Text.h>* 6-218
Figure 6-268: <Xm/TextF.h>* 6-218
Figure 6-269: <Xm/ToggleB.h>* 6-218
Figure 6-270: <Xm/ToggleBG.h>* 6-218
Figure 6-271: <Xm/VendorS.h>* 6-219
Figure 6-272: <Xm/VirtKeys.h>*, Part 1 of 2 6-219
Figure 6-273: <Xm/VirtKeys.h>*, Part 2 of 2 6-220
Figure 6-274: <Xm/Xm.h>*, Part 1 of 14 6-221
Figure 6-275: <Xm/Xm.h>*, Part 2 of 14 6-222
Figure 6-276: <Xm/Xm.h>*, Part 3 of 14 6-223
Figure 6-277: <Xm/Xm.h>*, Part 4 of 14 6-224
Figure 6-278: <Xm/Xm.h>*, Part 5 of 15 6-225
Figure 6-279: <Xm/Xm.h>*, Part 6 of 14 6-226
Figure 6-280: <Xm/Xm.h>*, Part 7 of 14 6-227
Figure 6-281: <Xm/Xm.h>*, Part 8 of 14 6-228
Figure 6-282: <Xm/Xm.h>*, Part 9 of 14 6-229
Figure 6-283: <Xm/Xm.h>*, Part 10 of 14 6-230
Figure 6-284: <Xm/Xm.h>*, Part 11 of 14 6-231
Figure 6-285: <Xm/Xm.h>*, Part 12 of 14 6-232
Figure 6-286: <Xm/Xm.h>*, Part 13 of 14 6-233
Figure 6-287: <Xm/Xm.h>*, Part 14 of 14 6-234
Figure 6-288: <Xm/XmStrDefs.h>*, Part 1 of 34 6-235
Figure 6-289: <Xm/XmStrDefs.h>*, Part 2 of 34 6-236
Figure 6-290: <Xm/XmStrDefs.h>*, Part 3 of 34 6-237

Figure 6-291: <Xm/XmStrDefs.h>*, Part 4 of 34 6-238
Figure 6-292: <Xm/XmStrDefs.h>*, Part 5 of 34 6-239
Figure 6-293: <Xm/XmStrDefs.h>*, Part 6 of 34 6-240
Figure 6-294: <Xm/XmStrDefs.h>*, Part 7 of 34 6-241
Figure 6-295: <Xm/XmStrDefs.h>*, Part 8 of 34 6-242
Figure 6-296: <Xm/XmStrDefs.h>*, Part 9 of 34 6-243
Figure 6-297: <Xm/XmStrDefs.h>*, Part 10 of 34 6-244
Figure 6-298: <Xm/XmStrDefs.h>*, Part 11 of 34 6-245
Figure 6-299: <Xm/XmStrDefs.h>*, Part 12 of 34 6-246
Figure 6-300: <Xm/XmStrDefs.h>*, Part 13 of 34 6-247
Figure 6-301: <Xm/XmStrDefs.h>*, Part 14 of 34 6-248
Figure 6-302: <Xm/XmStrDefs.h>*, Part 15 of 34 6-249
Figure 6-303: <Xm/XmStrDefs.h>*, Part 16 of 34 6-250
Figure 6-304: <Xm/XmStrDefs.h>*, Part 17 of 34 6-251
Figure 6-305: <Xm/XmStrDefs.h>*, Part 18 of 34 6-252
Figure 6-306: <Xm/XmStrDefs.h>*, Part 19 of 34 6-253
Figure 6-307: <Xm/XmStrDefs.h>*, Part 20 of 34 6-254
Figure 6-308: <Xm/XmStrDefs.h>*, Part 21 of 34 6-255
Figure 6-309: <Xm/XmStrDefs.h>*, Part 22 of 34 6-256
Figure 6-310: <Xm/XmStrDefs.h>*, Part 23 of 34 6-257
Figure 6-311: <Xm/XmStrDefs.h>*, Part 24 of 34 6-258
Figure 6-312: <Xm/XmStrDefs.h>*, Part 25 of 34 6-259
Figure 6-313: <Xm/XmStrDefs.h>*, Part 26 of 34 6-260
Figure 6-314: <Xm/XmStrDefs.h>*, Part 27 of 34 6-261
Figure 6-315: <Xm/XmStrDefs.h>*, Part 28 of 34 6-262
Figure 6-316: <Xm/XmStrDefs.h>*, Part 29 of 34 6-263
Figure 6-317: <Xm/XmStrDefs.h>*, Part 30 of 34 6-264
Figure 6-318: <Xm/XmStrDefs.h>*, Part 31 of 34 6-265
Figure 6-319: <Xm/XmStrDefs.h>*, Part 32 of 34 6-266
Figure 6-320: <Xm/XmStrDefs.h>*, Part 33 of 34 6-267
Figure 6-321: <Xm/XmStrDefs.h>*, Part 34 of 34 6-268
Figure 6-322: <netinet/in.h> 6-270
Figure 6-323: <netinet/ip.h> 6-271
Figure 6-324: <netinet/tcp.h> 6-271

1 INTRODUCTION

The Intel386 Architecture and the System V ABI 1-1

How to Use the Intel386 Architecture ABI Supplement 1-2
Evolution of the ABI Specification 1-2

DRAFT COPY
March 19, 1997
File: abi_386/Cchap1 (Delta 44.3)
386:adm.book:sum

The Intel386 Architecture and the System V ABI

The *System V Application Binary Interface*, or ABI, defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement the interfaces defined in the *System V Interface Definition, Edition 4*. This includes systems that have implemented UnixWare® 2.0. M

This document is a supplement to the generic *System V ABI*, and it contains information specific to System V implementations built on the Intel386 processor architecture. Together, these two specifications, the generic *System V ABI* and the *Intel386 Architecture System V ABI Supplement* (hereafter referred to as the *Intel386 ABI*), constitute a complete *System V Application Binary Interface* specification for systems that implement the processor architecture of the Intel386 microprocessors.

Note that, because the Intel486 and Pentium processor are compatible members of the Intel386 architecture, this *Intel386 ABI* also applies to any system built with the Intel486 or the Pentium processor chips. M M

How to Use the Intel386 Architecture ABI Supplement

This document is a supplement to the generic *System V ABI* and contains information referenced in the generic specification that may differ when System V is implemented on different processors. Therefore, the generic ABI is the prime reference document, and this supplement is provided to fill gaps in that specification.

As with the *System V ABI*, this specification references other publicly-available reference documents, especially the Intel *80386 Programmer's Reference Manual*. All the information referenced by this supplement should be considered part of this specification, and just as binding as the requirements and data explicitly included here.

Evolution of the ABI Specification

The *System V Application Binary Interface* will evolve over time to address new technology and market requirements, and will be reissued at intervals of approximately three years. Each new edition of the specification is likely to contain extensions and additions that will increase the potential capabilities of applications that are written to conform to the ABI.

As with the *System V Interface Definition*, the ABI will implement **Level 1** and **Level 2** support for its constituent parts. **Level 1** support indicates that a portion of the specification will continue to be supported indefinitely, while **Level 2** support means that a portion of the specification may be withdrawn or altered after the next edition of the ABI is made available. That is, a portion of the specification moved to **Level 2** support in an edition of the ABI specification will remain in effect at least until the following edition of the specification is published.

These **Level 1** and **Level 2** classifications and qualifications apply to this Supplement, as well as to the generic specification. All components of the ABI and of this supplement have **Level 1** support unless they are explicitly labelled as **Level 2**.

The following documents may be of interest to the reader of this specification:

- *i486 MICROPROCESSOR Programmer's Reference Manual* (Intel Literature order number 240486)

- *80386 Programmer's Reference Manual* (Intel Literature order number 230985)
- *80387 Programmer's Reference Manual* (Intel Literature order number 231917)
- *UnixWare® 2.0 Command Reference (a-l)*
- *UnixWare® 2.0 Command Reference (m-z)*
- *UnixWare® 2.0 Operating System API Reference: System Calls*
- *UnixWare® 2.0 Operating System API Reference: Library Functions*
- *UnixWare® 2.0 System Administration: Volumes I and II*
- *System V Interface Definition, Edition 4*

NOTE

Diffmarkings have been retained in the text of this book to indicate in which revisions of System V certain modifications were made to the *ABI*.

A "G" character in the right hand margin indicates a change in the *ABI* made in UNIX System V Release 4.2.

A "M" character in the right hand margin indicates a change in the *ABI* made in UnixWare® 2.0. M

DRAFT COPY
March 19, 1997
File: abi_386/chap1 (Delta 44.5)
386:adm.book:sum

Page: 17

2 SOFTWARE INSTALLATION

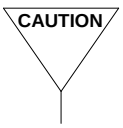
Software Distribution Formats 2-1
Physical Distribution Media 2-1
File System Formats 2-1
- s5 File System 2-1
- UFS File System 2-3

Software Distribution Formats

Physical Distribution Media

Approved media for physical distribution of ABI-conforming software are listed below. Inclusion of a particular medium on this list does not require an ABI-conforming system to accept that medium. For example, a conforming system may install all software through its network connection and accept none of the listed media.

- 1.44MB 3 1/2" floppy disk: quad-density, double-sided, 80 tracks/side, 18 sectors/track, 512 bytes/sector.
- 1.2MB 5 1/4" floppy disk: quad-density, double-sided, 80 tracks/side, 15 sectors/track, 512 bytes/sector.
- 360KB 5 1/4" floppy disk: double-density, double-sided, 40 tracks/side, 9 sectors/track, 512 bytes/sector.
- 60 MB quarter-inch cartridge tape in QIC-24 format. G
- CD-ROM optical disks. G
- 150 MB quarter-inch tape.



The use of 360KB 5 1/4" floppy disk, and 60 MB quarter inch cartridge tape as media for application distribution is moved to Level 2 as of January 1, 1993. G

File System Formats

Every file system storage volume must conform to a supported format. Two formats are supported: s5 and ufs.

s5 File System

The first physical block on the medium should be empty, and the second contains the device's *superblock*. The third contains an inode list, and remaining blocks on the device contain data. The *superblock* has the following format:

```
#define NICFREE             50
#define NICINOD             100

struct filsys {
         u_short   s_isize;
         daddr_t   s_fsize;
         short             s_nfree;
         daddr_t   s_free[NICFREE];
         short             s_ninode;
         ushort_t  s_inode[NICINOD];
         char              s_flock;
         char              s_ilock;
         char              s_fmod;
         char              s_ronly;
         time_t            s_time;
         short             s_dinfo[4];
         daddr_t   s_tfree;
         ushort_t  s_tinode;
         char              s_fname[6];
         char              s_fpack[6];
         long              s_fill[12];
         long              s_state;
         long              s_magic;
         long              s_type;
};

#define FsMAGIC    0xfd187e20

#define Fs1b               1
#define Fs2b               2
#define Fs4b               3

#define FsOKAY     0x7c269d38
#define FsACTIVE   0x5e72d81a
#define FsBAD              0xcb096f43
#define FsBADBLK   0xbadbc14b
```

s_type indicates the file system type. Currently, three types of file systems are supported: the original 512-byte logical block, the 1024-byte logical block, and the 2048-byte logical block. *s_magic* is used to distinguish the original 512-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, `fsMAGIC`, the type is assumed to be `fs1b`, otherwise the *s_type* field is used.

s_state indicates the state of the file system. A cleanly unmounted, undamaged file system is indicated by the `FsOKAY` state. After a file system has been mounted for update, the state changes to `FsACTIVE`.

s_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is *s_isize*–2 blocks long.

s_fsize is the first block not potentially available for allocation to a file.

The free list for each volume is maintained as follows. The *s_free* array contains up to 49 numbers of free blocks. *s_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain.

s_tfree is the total free blocks available in the file system.

s_ninode is the number of free i-numbers in the *s_inode* array.

s_tinode is the total free i-nodes available in the file system.

s_flock and *s_ilock* are flags maintained in the core copy of the file system. *s_fmod* is a flag that indicates that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

s_ronly is a read-only flag to indicate write-protection.

s_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT).

s_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. I-node 1 is reserved for future use. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file.

UFS File System

In the UFS file system, the first physical block on the device should be empty, and the second contains the *superblock* for the file system. Remaining blocks contain data.

The ufs *superblock* contains an *fs* data structure. This structure, and other relevant data objects are defined below.

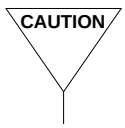
```
struct csum {
         long          cs_ndir;
         long          cs_nbfree;
         long          cs_nifree;
         long          cs_nffree;
};

struct fs {
         struct fs     *fs_link;
         struct fs     *fs_rlink;
         daddr_t       fs_sblkno;
         daddr_t       fs_cblkno;
         daddr_t       fs_iblkno;
         daddr_t       fs_dblkno;
         long          fs_cgoffset;
         long          fs_cgmask;
         time_t        fs_time;
         long          fs_size;
         long          fs_dsize;
         long          fs_ncg;
         long          fs_bsize;
         long          fs_fsize;
         long          fs_frag;
         long          fs_minfree;
         long          fs_rotdelay;
         long          fs_rps;
         long          fs_bmask;
         long          fs_fmask;
         long          fs_bshift;
         long          fs_fshift;
         long          fs_maxcontig;
         long          fs_maxbpg;
         long          fs_fragshift;
         long          fs_fsbtodb;
         long          fs_sbsize;
         long          fs_csmask;
         long          fs_csshift;
         long          fs_nindir;
         long          fs_inopb;
         long          fs_nspf;
         long          fs_optim;
         long          fs_state;
         long          fs_sparecon[2];
         long          fs_id[2];
         daddr_t       fs_csaddr;
         long          fs_cssize;
         long          fs_cgsize;
         long          fs_ntrak;
         long          fs_nsect;
         long          fs_spc;
         long          fs_ncyl;
```

(continued on next page)

```
        long          fs_cpg;
        long          fs_ipg;
        long          fs_fpg;
        struct csum   fs_cstotal;
        char          fs_fmod;
        char          fs_clean;
        char          fs_ronly;
        char          fs_flags;
        char          fs_fsmnt[MAXMNTLEN];
        long          fs_cgrotor;
        struct csum   *fs_csp[MAXCSBUFS];
        long          fs_cpc;
        short         fs_postbl[MAXCPG][NRPOS];
        long          fs_magic;
        u_char        fs_rotbl[1];
};

struct cg {
        struct cg     *cg_link;
        struct cg     *cg_rlink;
        time_t        cg_time;
        long          cg_cgx;
        short         cg_ncyl;
        short         cg_niblk;
        long          cg_ndblk;
        struct csum   cg_cs;
        long          cg_rotor;
        long          cg_frotor;
        long          cg_irotor;
        long          cg_frsum[MAXFRAG];
        long          cg_btot[MAXCPG];
        short         cg_b[MAXCPG][NRPOS];
        char          cg_iused[MAXIPG/NBBY];
        long          cg_magic;
        u_char        cg_free[1];
};

#define FS_MAGIC       0x011954
#define BBSIZE         8192
#define SBSIZE         8192
#define BBLOCK         ((daddr_t)(0))
#define SBLOCK         ((daddr_t)(BBLOCK + BBSIZE / DEV_BSIZE))
#define UFSROOTINO     ((ino_t)2)
#define LOSTFOUNDINO   (UFSROOTINO + 1)
#define NRPOS          8
#define MAXIPG         2048
#define MINBSIZE       4096
#define MAXCPG         32
#define MAXMNTLEN      512
#define MAXCSBUFS      32
#define FS_OPTTIME     0
```

(continued on next page)

```
#define FS_OPTSPACE     1
#define MAXBPC          (SBSIZE - sizeof (struct fs))
#define CG_MAGIC        0x090255
```



The distribution of software in filesystem format is Level 2 as of January 1, 1993. G

3 LOW-LEVEL SYSTEM INFORMATION

Machine Interface 3-1
Processor Architecture 3-1
Data Representation 3-1
- Fundamental Types 3-2
- Aggregates and Unions 3-3
- Bit-Fields 3-6

Function Calling Sequence 3-9
Registers and the Stack Frame 3-9
Functions Returning Scalars or No Value 3-12
Functions Returning Structures or Unions 3-14
Integral and Pointer Arguments 3-17
Floating-Point Arguments 3-17
Structure and Union Arguments 3-18

Operating System Interface 3-20
Virtual Address Space 3-20
- Page Size 3-20
- Virtual Address Assignments 3-20
- Managing the Process Stack 3-22
- Coding Guidelines 3-23

Processor Execution Modes 3-24
Exception Interface 3-24
- Hardware Exception Types 3-24
- Software Trap Types 3-25

Process Initialization 3-26
- Special Registers 3-26
- Process Stack and Registers 3-28

Coding Examples 3-34
Code Model Overview 3-35
Position-Independent Function Prologue 3-36
Data Objects 3-37
Function Calls 3-39
Branching 3-41
C Stack Frame 3-43
Variable Argument List 3-44
Allocating Stack Space Dynamically 3-44

Machine Interface

Processor Architecture

The Intel *80386 Programmer's Reference Manual* (Intel Literature order number 230985) and the Intel *80387 Programmer's Reference Manual* (Intel Literature order number 231917) together define the processor architecture. The architecture of the combined Intel386/Intel 387 processors is hereafter referred to as the Intel386 architecture. Programs intended to execute directly on the processor use the instruction set, instruction encodings, and instruction semantics of the architecture. Three points deserve explicit mention.

- A program may assume all documented instructions exist.
- A program may assume all documented instructions work.
- A program may use only the instructions defined by the architecture.

In other words, *from a program's perspective*, the execution environment provides a complete and working implementation of the Intel386 architecture.

This does not imply that the underlying implementation provides all instructions in hardware, only that the instructions perform the specified operations and produce the specified results. The ABI neither places performance constraints on systems nor specifies what instructions must be implemented in hardware. A software emulation of the architecture could conform to the ABI.

Some processors might support the Intel386 architecture as a subset, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to the Intel386 ABI. Executing those programs on machines without the additional capabilities gives undefined behavior.

Data Representation

Within this specification, the term *halfword* refers to a 16-bit object, the term *word* refers to a 32-bit object, and the term *doubleword* refers to a 64-bit object.

Fundamental Types

Figure 3-1 shows the correspondence between ANSI C's scalar types and the processor's.

Figure 3-1: Scalar Types

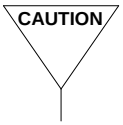
Type	C	sizeof	Alignment (bytes)	Intel386 Architecture
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed word

	`unsigned int` `unsigned long`	4	4	unsigned word
Pointer	*any-type* `*` *any-type* `(*)()`	4	4	unsigned word
Floating-point	`float`	4	4	single-precision (IEEE)
	`double`	8	4	double-precision (IEEE)
	`long double`	12	4	extended-precision (IEEE)

NOTE

The Intel386 architecture does not require doubleword alignment for double-precision values. Nevertheless, for data structure compatibility with other Intel architectures, compilers may provide a method to align double-precision values on doubleword boundaries.



A compiler that provides the doubleword alignment mentioned above can generate code (data structures and function calling sequences) that do not conform to the Intel386 ABI. Programs built with the doubleword alignment facility can thus violate conformance to the Intel386 ABI. See ''Aggregates and Unions'' below and ''Function Calling Sequence'' later in this chapter for more information.

A null pointer (for all types) has the value zero.

The Intel386 architecture does not require all data access to be properly aligned. For example, double-precision values occupy 1 doubleword (8-bytes), and their natural alignment is a word boundary, meaning their addresses are multiples of 4. Compilers should allocate independent data objects with the proper alignment; examples include global arrays of double-precision variables, FORTRAN `COMMON` blocks, and unconstrained stack objects. However, some language facilities (such as FORTRAN `EQUIVALENCE` statements) may create objects with only byte alignment. Consequently, arbitrary data accesses, such as pointers dereference or reference arguments, might or might not be properly aligned. Accessing misaligned data will be slower than accessing properly aligned data, but otherwise there is no difference.

Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, is always a multiple of the object's alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined. G

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

NOTE

ABI conformant code may not read or modify anything marked reserved or padding. M

In the following examples, members' byte offsets appear in the upper right corners.

Figure 3-2: Structure Smaller Than a Word

```
struct {
    char   c;
};
```

Byte aligned, `sizeof` is 1

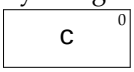


Figure 3-3: No Padding

```
struct {
    char   c;
    char   d;
    short  s;
    long   n;
};
```

Word aligned, `sizeof` is 8

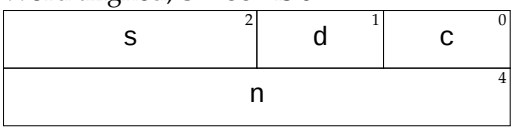


Figure 3-4: Internal Padding

```
struct {
    char   c;
    short  s;
};
```

Halfword aligned, `sizeof` is 4

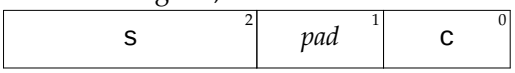
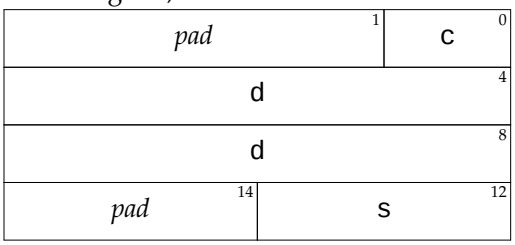


Figure 3-5: Internal and Tail Padding

```
struct {
    char   c;
    double d;
    short  s;
};
```

Word aligned, `sizeof` is 16



NOTE

The Intel386 architecture does not require doubleword alignment for double-precision values. Nevertheless, for data structure compatibility with other Intel architectures, compilers may provide a method to align double-precision values on doubleword boundaries.

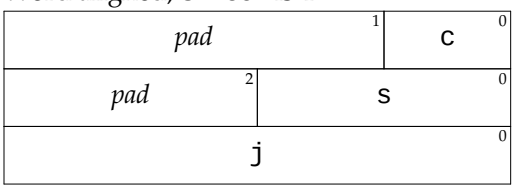
CAUTION

A compiler that provides the doubleword alignment mentioned above would arrange the preceding structure differently. Programs built with the double-word alignment facility would not conform to the Intel386 ABI, and they would not be data-compatible with conforming Intel386 programs.

Figure 3-6: `union` **Allocation**

```
union {
    char   c;
    short  s;
    int    j;
};
```

Word aligned, `sizeof` is 4



Bit-Fields

C `struct` and `union` definitions may have *bit-fields*, which define integral objects with a specified number of bits.

Figure 3-7: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed char`	1 to 8	-2^{w-1} to $2^{w-1}-1$
`char`		0 to $2^{w}-1$
`unsigned char`		0 to $2^{w}-1$
`signed short`	1 to 16	-2^{w-1} to $2^{w-1}-1$
`short`		0 to $2^{w}-1$
`unsigned short`		0 to $2^{w}-1$
`signed int`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`int`		0 to $2^{w}-1$
`enum`		0 to $2^{w}-1$
`unsigned int`		0 to $2^{w}-1$
`signed long`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`long`		0 to $2^{w}-1$
`unsigned long`		0 to $2^{w}-1$

''Plain'' bit-fields (that is, those neither `signed` nor `unsigned`) always have non-negative values. Although they may have type `char`, `short`, `int`, or `long` (which can have negative values), these bit-fields have the same range as a bit-field of the same size with the corresponding `unsigned` type. Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions:

- Bit-fields are allocated from right to left (least to most significant).
- A bit-field must entirely reside in a storage unit appropriate for its declared type. Thus a bit-field never crosses its unit boundary.
- Bit-fields may share a storage unit with other `struct/union` members, including members that are not bit-fields. Of course, `struct` members occupy different parts of the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union, although individual bit-fields' member offsets obey the alignment constraints.

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

The following examples show `struct` and `union` members' byte offsets in the upper right corners; bit numbers appear in the lower corners.

Figure 3-8: Bit Numbering

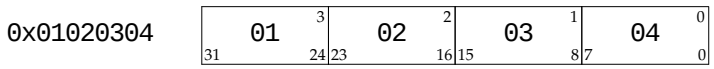


Figure 3-9: Right-to-Left Allocation

```
struct {
    int    j:5;
    int    k:6;
    int    m:7;
};
```

Word aligned, `sizeof` is 4

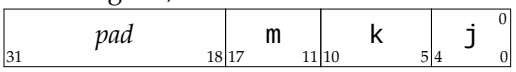


Figure 3-10: Boundary Alignment

```
struct {
    short  s:9;
    int    j:9;
    char   c;
    short  t:9;
    short  u:9;
    char   d;
};
```

Word aligned, `sizeof` is 12

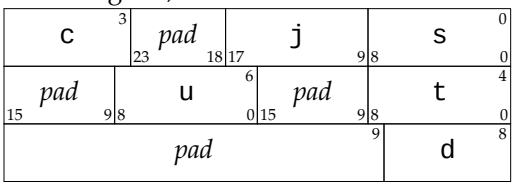


Figure 3-11: Storage Unit Sharing

```
struct {
    char   c;
    short  s:8;
};
```

Halfword aligned, `sizeof` is 2

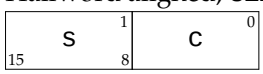


Figure 3-12: `union` **Allocation**

```
union {
    char   c;
    short  s:8;
};
```

Halfword aligned, `sizeof` is 2

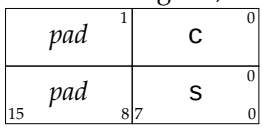
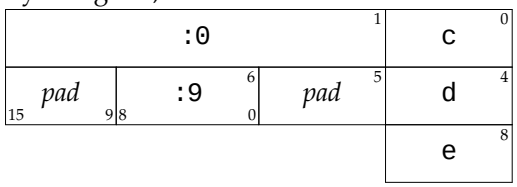


Figure 3-13: Unnamed Bit-Fields

```
struct {
    char   c;
    int    :0;
    char   d;
    short  :9;
    char   e;
    char   :0;
};
```

Byte aligned, `sizeof` is 9



As the examples show, `int` bit-fields (including `signed` and `unsigned`) pack more densely than smaller base types. One can use `char` and `short` bit-fields to force particular alignments, but `int` is generally more efficient.

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, parameter passing, and so on. The system libraries described in Chapter 6 require this calling sequence.

NOTE

The standard calling sequence *requirements* apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nonetheless, it is recommended that all functions use the standard calling sequence when possible.

NOTE

C programs follow the conventions given here. For specific information on the implementation of C, see ''Coding Examples'' in this chapter.

Registers and the Stack Frame

The Intel386 architecture provides a number of registers. All the integer registers and all the floating-point registers are global to all procedures in a running program.

Brief register descriptions appear in Figure 3-14 more complete information appears later.

Figure 3-14: Processor Registers

Type	Name	Usage
General	%eax	Return value
	%edx	Dividend register (divide operations)
	%ecx	Count register (shift and string operations)
	%ebx	Local register variable
	%ebp	Stack frame pointer (optional) G
	%esi	Local register variable
	%edi	Local register variable
	%esp	Stack pointer
Floating-point	%st(0)	floating-point stack top, return value
	%st(1)	floating-point next to stack top
	. . .	
	%st(7)	floating-point stack bottom

In addition to registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Figure 3-15 shows the stack frame organization.

Figure 3-15: Standard Stack Frame

Position	Contents	Frame	
4*n*+8(%ebp)	argument word *n*	Previous	*High addresses*
	. . .		
8(%ebp)	argument word 0		
4(%ebp)	return address	Current	
0(%ebp)	previous %ebp (optional)		
-4(%ebp)	unspecified		
	. . .		
0(%esp)	variable size		*Low addresses*

Several key points about the stack frame deserve mention.

- The stack is word aligned. Although the architecture does not require any alignment of the stack, software convention and the operating system requires that the stack be aligned on a word boundary. G G

- Argument words are pushed onto the stack in reverse order (that is, the rightmost argument in C call syntax has the highest address), preserving the stack's word alignment. All incoming arguments appear on the stack, residing in the stack frame of the caller.
- An argument's size is increased, if necessary, to make it a multiple of words. This may require tail padding, depending on the size of the argument.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses the ''unspecified'' area of the standard stack frame.

All registers on the Intel386 are global and thus visible to both a calling and a called function. Registers `%ebp`, `%ebx`, `%edi`, `%esi`, and `%esp` ''belong'' to the calling function. In other words, a called function must preserve these registers' values for its caller. Remaining registers ''belong'' to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

Some registers have assigned roles in the standard calling sequence:

`%esp`	The *stack pointer* holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. At all times, the stack pointer should point to a word-aligned area.
`%ebp`	The *frame pointer* optionally holds a base address for the current stack frame. Consequently, a function has registers pointing to both ends of its frame. Incoming arguments reside in the previous frame, referenced as positive offsets from `%ebp`, while local variables reside in the current frame, referenced as negative offsets from `%ebp`. A function must preserve this register's value for its caller. G
`%eax`	*Integral and pointer return values* appear in `%eax`. A function that returns a `struct` or `union` value places the address of the result in `%eax`. Otherwise this is a scratch register.
`%ebx`	As described below, this register serves as the *global offset table base register* for position-independent code. For absolute code, `%ebx` serves as a local register and has no specified role in the function calling sequence. In either case, a function must preserve the register value for the caller.
`%esi` and `%edi`	These *local registers* have no specified role in the function calling sequence. A function must preserve their values for the caller.

`%ecx` and `%edx` *Scratch registers* have no specified role in the standard calling sequence. Functions do not have to preserve their values for the caller.

`%st(0)` *Floating-point return values* appear on the top of the floating-point register stack; there is no difference in the representation of single- or double-precision values in floating-point registers. If the function does not return a floating-point value, then this register must be empty. This register must be empty before entry to a function. G

`%st(1)` through `%st(7)`
Floating-point scratch registers have no specified role in the standard calling sequence. These registers must be empty before entry and upon exit from a function.

EFLAGS The *flags register* contains the system flags, such as the direction flag and the carry flag. The direction flag must be set to the ''forward'' (that is, zero) direction before entry and upon exit from a function. Other user flags have no specified role in the standard calling sequence and are not preserved.

Floating-Point Control Word
The Intel387 *control word* contains the floating-point flags, such as the rounding mode and exception masking. *

Signals can interrupt processes [see `signal`(BA_OS)]. Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without the danger of signal handlers changing their values.

Functions Returning Scalars or No Value

A function that returns an integral or pointer value places its result in register `%eax`.

A floating-point return value appears on the top of the Intel387 register stack. The caller then must remove the value from the Intel387 stack, even if it doesn't use the value. Failure of either side to meet its obligations leads to undefined program behavior. The standard calling sequence does not include any method to detect such failures nor to detect return value type mismatches. Therefore the user must declare all functions properly. There is no difference in the representation of

single-, double- or extended-precision values in floating-point registers.

Functions that return no value (also called procedures or `void` functions) put no particular value in any register.

A `call` instruction pushes the address of the next instruction (the return address) onto the stack. The `ret` instruction pops the address off the stack and effectively continues execution at the next instruction after the `call` instruction. A function that returns a scalar or no value must preserve the caller's registers as described earlier. Additionally, the called function must remove the return address from the stack, leaving the stack pointer (`%esp`) with the value it had before the `call` instruction was executed.

To illustrate, the following function prologue allocates 80 bytes of local stack space and saves the local registers `%ebx`, `%esi`, and `%edi`.

Figure 3-16: Function Prologue

```
prologue:
        pushl  %ebp          / save frame pointer
        movl   %esp, %ebp    / set new frame pointer
        subl   $80, %esp     / allocate stack space
        pushl  %edi          / save local register
        pushl  %esi          / save local register
        pushl  %ebx          / save local register
```

An epilogue for the example that restores the state for the caller. This example returns the value in `%edi` by moving it to `%eax`.

Figure 3-17: Function Epilogue

```
        movl   %edi, %eax    / set up return value
epilogue:
        popl   %ebx          / restore local register
        popl   %esi          / restore local register
        popl   %edi          / restore local register
        leave                / restore frame pointer
        ret                  / pop return address
```

NOTE Although some functions can be optimized to eliminate the save and restore of the frame pointer, the general case uses the standard prologue and epilogue.

Sections below describe where arguments appear on the stack. The examples are written as if the function prologue described above had been used.

Position-independent code uses the %ebx register to hold the address of the global offset table. If a function needs the global offset table's address, either directly or indirectly, it is responsible for computing the value. See ''Coding Examples'' later in this chapter and ''Dynamic Linking'' in Chapter 5 for more information.

Functions Returning Structures or Unions

If a function returns a structure or union, then the caller provides space for the return value and places its address on the stack as argument word zero. In effect, this address becomes a ''hidden'' first argument. Having the caller supply the return object's space allows re-entrancy.

NOTE Structures and unions in this context have fixed sizes. The ABI does not specify how to handle variable sized objects.

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

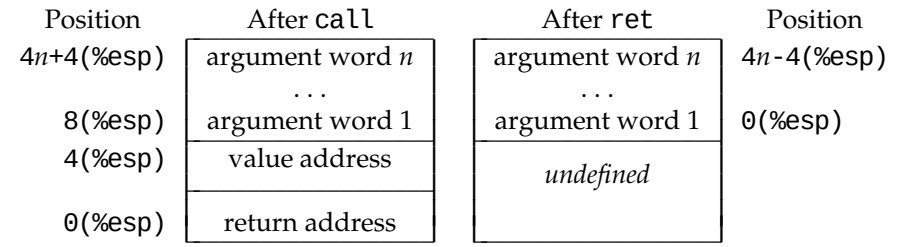
A function that returns a structure or union also sets `%eax` to the value of the original address of the caller's area before it returns. Thus when the caller receives control again, the address of the returned object resides in register `%eax` and can be used to access the object. Both the calling and the called functions must cooperate to pass the return value successfully:

- The calling function must supply space for the return value and pass its address in the stack frame;
- The called function must use the address from the frame and copy the return value to the object so supplied;
- The called function must remove this address from the stack before returning.

Failure of either side to meet its obligations leads to undefined program behavior. The standard function calling sequence does not include any method to detect such failures nor to detect structure and union type mismatches. Therefore the user must declare all functions properly.

Figure 3-18 illustrates the stack contents when the function receives control (after the `call` instruction) and when the calling function again receives control (after the `ret` instruction).

Figure 3-18: Stack Contents for Functions Returning `struct/union`



To illustrate, the following function prologue allocates 80 bytes of local stack space and saves the local registers `%ebx`, `%esi`, and `%edi`. Additionally, it removes the ''hidden'' argument from the stack and saves it in the highest word of the local stack frame.

Figure 3-19: Function Prologue (Returning `struct/union`)

```
prologue:
        popl   %eax            / pop return address
        xchgl  %eax, 0(%esp)   / swap return address
                                  / and return value address
        pushl  %ebp            / save frame pointer
        movl   %esp, %ebp      / set new frame pointer
        subl   $80, %esp       / allocate local space
        pushl  %edi            / save local register
        pushl  %esi            / save local register
        pushl  %ebx            / save local register
        movl   %eax, -4(%ebp) / save return value address
```

An epilogue for the example that restores the state for the caller.

Figure 3-20: Function Epilogue

```
epilogue:
        movl   -4(%ebp), %eax      / set up return value
        popl   %ebx                / restore local register
        popl   %esi                / restore local register
        popl   %edi                / restore local register
        leave                     / restore frame pointer
        ret                       / pop return address
```


DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

NOTE

Although some functions can be optimized to eliminate the save and restore of the frame pointer, the general case uses the standard prologue and epilogue.

Sections below describe where arguments appear on the stack. The examples are written as if the function prologue described above had been used.

Position-independent code uses the `%ebx` register to hold the address of the global offset table. If a function needs the global offset table's address, either directly or indirectly, it is responsible for computing the value. See ''Coding Examples'' later in this chapter and ''Dynamic Linking'' in Chapter 5 for more information.

Integral and Pointer Arguments

As mentioned, a function receives all its arguments through the stack; the last argument is pushed first. In the standard calling sequence, the first argument is at offset `8(%ebp)`, the second argument is at offset `12(%ebp)`, and so on. Functions pass all integer-valued arguments as words, expanding or padding signed or unsigned bytes and halfwords as needed.

Figure 3-21: Integral and Pointer Arguments

Call	Argument	Stack address
	1	8(%ebp)
g(1, 2, 3,	2	12(%ebp)
(void *)0);	3	16(%ebp)
	(void *)0	20(%ebp)

Floating-Point Arguments

The stack also holds floating-point arguments: single-precision values use one word, double-precision use two, and extended-precision use three. See ''Coding Examples'' for information about floating-point arguments and variable argument lists. The example below uses only double-precision arguments. Single- and extended-precision arguments behave as specified above.

Figure 3-22: Floating-Point Arguments

Call	Argument	Stack address
`h(1.414, 1, 2.998e10);`	word 0, `1.414`	`8(%ebp)`
	word 1, `1.414`	`12(%ebp)`
	`1`	`16(%ebp)`
	word 0, `2.998e10`	`20(%ebp)`
	word 1, `2.998e10`	`24(%ebp)`

NOTE

The Intel386 architecture does not require doubleword alignment for double-precision values. Nevertheless, for data structure compatibility with other Intel architectures, compilers may provide a method to align double-precision values on doubleword boundaries.

CAUTION

A compiler that provides the doubleword alignment mentioned above would have to maintain doubleword alignment for the stack. Moreover, the arguments in the preceding example would appear in different positions. Programs built with the doubleword alignment facility would not conform to the Intel386 ABI, and their function calling sequence would not be compatible with conforming Intel386 programs.

Structure and Union Arguments

As described in the data representation section, structures and unions can have byte, halfword, or word alignment, depending on the constituents. An argument's size is increased, if necessary, to make it a multiple of words. This may require tail padding, depending on the size of the argument. To ensure that data in the stack is properly aligned, the stack pointer should always point to a word boundary. Structure and union arguments are pushed onto the stack in the same manner as integral arguments, described above. This provides call-by-value semantics, letting the called function modify its arguments without affecting the calling function's object.

Figure 3-23: Structure and Union Arguments

Call	Argument	Callee
`i(1, s);`	`1`	`8(%ebp)`
	word 0, `s`	`12(%ebp)`
	word 1, `s`	`16(%ebp)`
	...	...

Operating System Interface

Virtual Address Space

Processes execute in a 32-bit virtual address space. Memory management translates virtual addresses to physical addresses, hiding physical addressing and letting a process run anywhere in the system's real memory. Processes typically begin with three logical segments, commonly called text, data, and stack. As Chapter 5 describes, dynamic linking creates more segments during execution, and a process can create additional segments for itself with system services.

Page Size

Memory is organized by pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another, depending on the processor, memory management unit and system configuration. Processes may call `sysconf`(BA_OS) to determine the system's current page size. *

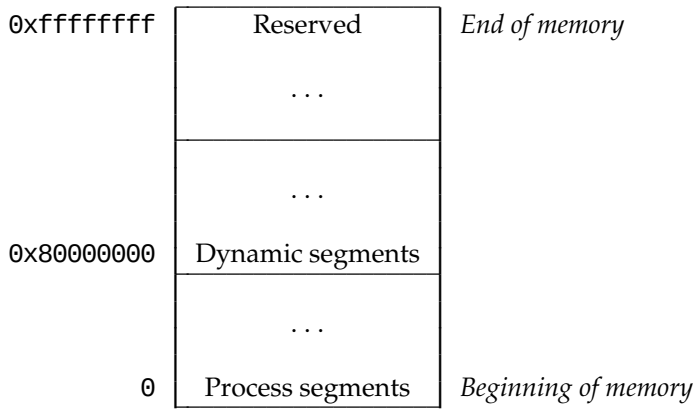
Virtual Address Assignments

Conceptually, processes have the full 32-bit address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration-dependent amount of virtual space.
- The system reserves a configuration dependent amount of space per process. G
- A process whose size exceeds the system's available, combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amounts.

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

Figure 3-24: Virtual Address Configuration



CAUTION

Programs that dereference null pointers are erroneous. although an implementation is not obliged to detect such erroneous behavior. Such programs may or may not fail on a particular system. To enhance portability, programmers are strongly cautioned not to rely on this behavior. G G

Process segments
: Processes' loadable segments and stack may begin at `0`. The exact addresses depend on the executable file format [see further information below and in Chapters 4 and 5]. Processes can control the amount of virtual memory allotted for stack space, as described below.

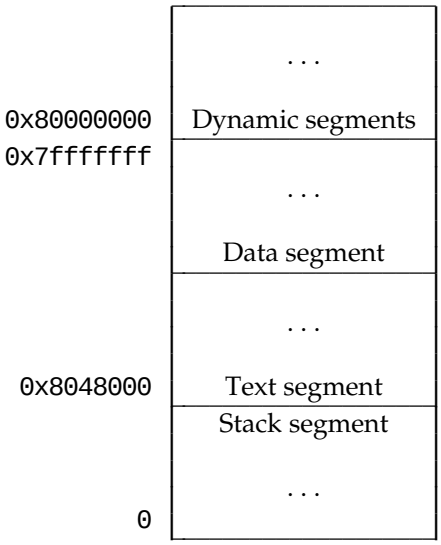
Dynamic segments
: A process's dynamic segments reside below the reserved area.

Reserved
: A reserved area resides at the top of virtual space.

As the figure shows, the system reserves the high end of virtual address space, with a process's dynamic segments below that. Although the exact boundary between the reserved area and a process depends on the system's configuration, the reserved area shall not consume more than 1 GB of the address space. Thus the user virtual address range has a minimum upper bound of `0xc0000000`. Individual systems may reserve less space, increasing processes' virtual memory range.

Although applications may control their memory assignments, the typical arrangement appears below.

Figure 3-25: Conventional Segment Arrangements



The process's text segment resides at `0x8048000`. The data segment follows immediately, and dynamic segments occupy the higher range. When applications let the system choose addresses for dynamic segments (including shared object segments), it chooses high addresses. This leaves the ''middle'' of the address spectrum available for dynamic memory allocation with facilities such as `malloc`(BA_OS). Processes should *not* depend on finding their dynamic segments at particular virtual addresses. Facilities exist to let the system choose dynamic segment virtual addresses. The stack resides immediately below the text segment, growing toward lower addresses. This arrangement provides a little over 128 MB for the stack and about 2 GB for text and data.

Managing the Process Stack

Section ''Process Initialization'' in this chapter describes the initial stack contents. Stack addresses can change from one system to the next—even from one process execution to the next on the same system. Processes, therefore, should *not* depend on finding their stack at a particular virtual address.

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

A tunable configuration parameter controls the system maximum stack size. A process also can use `setrlimit`(BA_OS), to set its own maximum stack size, up to the system limit. On the Intel386, the stack segment has read and write permissions.

Coding Guidelines

Operating system facilities, such as `mmap`(KE_OS), allow a process to establish address mappings in two ways. First, the program can let the system choose an address. Second, the program can force the system to use an address the program supplies. This second alternative can cause application portability problems, because the requested address might not always be available. Differences in virtual address space can be particularly troublesome between different architectures, but the same problems can arise within a single architecture.

Processes' address spaces typically have three segment areas that can change size from one execution to the next: the stack [through `setrlimit`(BA_OS)], the data segment [through `malloc`(BA_OS)], and the dynamic segment area [through `mmap`(KE_OS)]. Changes in one area may affect the virtual addresses available for another. Consequently, an address that is available in one process execution might not be available in the next. A program that used `mmap`(KE_OS) to request a mapping at a specific address thus could appear to work in some environments and fail in others. For this reason, programs that wish to establish a mapping in their address space should let the system choose the address.

Despite these warnings about requesting specific addresses, the facility is both useful and can be used in a controlled manner. For example, a multiprocess application might map several files into the address space of each process and build relative pointers among the files' data. This could be done by having each process ask for a certain amount of memory at an address chosen by the system. After each process receives its own, private address from the system, it would map the desired files into memory, at specific addresses within the original area. This collection of mappings could be at different addresses in each process but their *relative* positions would be fixed. Without the ability to ask for specific addresses, the application could not build shared data structures, because the relative positions for files in each process would be unpredictable.

Processor Execution Modes

Four execution modes exist in the Intel386 architecture: ring 3 (or user mode) and three privileged rings. User processes run in user mode ring (the least privileged). The operating system kernel runs in a privileged mode ring, although the ABI does not specify which one. A program executes the `lcall` instruction through a system call gate to change execution modes, and thus the `lcall` instruction provides the low-level interface to system calls. For the Intel386, one low-level interface is defined: `_exit`(BA_OS).

To ensure a process has a way to terminate itself, the system treats `_exit` as a special case. The ABI does not specify the implementation of other system services. Instead, programs should use the system libraries that Chapter 6 describes. Programs with other embedded `lcall` instructions do not conform to the ABI.

Figure 3-26: `_exit` System Trap

```
        .globl _exit
_exit:
        movl   $1, %eax
        lcall  $7, $0
```

Exception Interface

As the Intel386 architecture manuals describe, the processor changes mode to handle *exceptions*, which may be synchronous, floating-point/coprocessor, or asynchronous. Synchronous and floating-point/coprocessor exceptions, being caused by instruction execution, can be explicitly generated by a process. This section, therefore, specifies those exception types with defined behavior. The Intel386 architecture classifies exceptions as *faults*, *traps*, and *aborts*. See the Intel *80386 Programmer's Reference Manual* for more information about their differences.

Hardware Exception Types

The operating system defines the following correspondence between hardware exceptions and the signals specified by `signal`(BA_OS).

Figure 3-27: Hardware Exceptions and Signals

Number	Exception Name	Signal
0	divide error fault	SIGFPE
1	single step trap/fault	SIGTRAP
2	nonmaskable interrupt	none
3	breakpoint trap	SIGTRAP
4	overflow trap	SIGSEGV
5	bounds check fault	SIGSEGV
6	invalid opcode fault	SIGILL
7	no coprocessor fault	SIGFPE
8	double fault abort	none
9	coprocessor overrun abort	SIGSEGV
10	invalid TSS fault	none
11	segment not present fault	none
12	stack exception fault	SIGSEGV
13	general protection fault/abort	SIGSEGV
14	page fault	SIGSEGV
15	(reserved)	
16	coprocessor error fault	SIGFPE
other	(unspecified)	SIGILL

Floating-point instructions exist in the architecture, but they may be implemented either in hardware (via the Intel387 chip) or in software (via the Intel387 emulator). In the case of ''no coprocessor'' exception, if the Intel387 emulator is configured into the kernel, the process receives no signal. Instead, the system intercepts the exception, emulates the instruction, and returns control to the process. A process receives SIGFPE for the ''no coprocessor'' exception only when the indicated floating-point instruction is illegal (invalid operands, and so on).

Software Trap Types

Because the int instruction generates traps, some hardware exceptions can be generated by software. However, the int instruction generates only traps and not faults; so it is not possible to match the exact hardware generated faults in software.

Process Initialization

This section describes the machine state that exec(BA_OS) creates for ''infant'' processes, including argument passing, register usage, stack frame layout, and so on. Programming language systems use this initial program state to establish a standard environment for their application programs. As an example, a C program begins executing at a function named main, conventionally declared in the following way.

Figure 3-28: Declaration for main

```
extern int main(int argc, char *argv[], char *envp[]);
```

Briefly, argc is a non-negative argument count; argv is an array of argument strings, with argv[argc]==0; and envp is an array of environment strings, also terminated by a null pointer.

Although this section does not describe C program initialization, it gives the information necessary to implement the call to main or to the entry point for a program in any other language.

Special Registers

As the Intel386 architecture defines, several state registers control and monitor the processor: the Machine Status Word register (MSW, also known as register %cr0), EFLAGS register, the floating-point status register, and the floating-point control register. Application programs cannot access the full EFLAGS register directly; because they run in the processor's *user mode*, and the instructions to write some of the bits of the EFLAGS register are privileged. Nonetheless, a program has access to many of the flags in the EFLAGS register. Flags identified with an ''*'' below are not modifiable by a user mode process, they either have unspecified values or do not affect user program behavior. At process initialization, the EFLAGS register contains the following values.

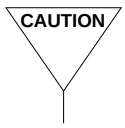
Figure 3-29: EFLAGS Register Fields

Flag	Value	Note
CF	unspecified	Carry flag
PF	unspecified	Parity flag
AF	unspecified	Auxiliary carry flag
ZF	unspecified	Zero flag
SF	unspecified	Sign flag
TF	unspecified	Trap flag
IF*	unspecified	Interrupt enable
DF	0	Direction flag low to high
OF	unspecified	Overflow flag
IOPL*	unspecified	I/O privilege level
NT*	unspecified	Nested task
RF*	unspecified	Resume flag
VM*	unspecified	Virtual 8086 mode

The Intel386 architecture defines floating-point instructions, and those instructions work whether the processor has a hardware floating-point unit or not. (A system may provide hardware or software floating-point facilities.) Consequently, the contents of the MSW register is not specified, letting the system set it according to the hardware configuration. In any case, however, the processor presents a working floating-point implementation, including the Intel387 status and control word registers with the following values at process initialization.

Figure 3-30: Floating-Point Control Word

Field	Value	Note	
IC	1	Affine infinity (for compatibility)	
RC	00	Round to nearest or even	
PC	11	53-bit (double precision)	G
PM	1	Precision masked	
UM	1	Underflow masked	
OM	1	Overflow	G
ZM	1	Zero divide	G
DM	1	Denormalized operand masked	
IM	1	Invalid operation	G



The initial floating-point state should be changed with care. In particular, many floating-point routines may produce undefined behavior if the precision control is set to less than 53 bits. The _fpstart routine (see Chapter 6) changes the precision control to 64 bits and sets all exceptions to be asked. This is the default state required for conformance to the ANSI C standard and to the IEEE 754 Floating-point standard. G G G G

Process Stack and Registers

When a process receives control, its stack holds the arguments and environment from exec(BA_OS).

Figure 3-31: Initial Process Stack

	Unspecified	*High addresses*
	Information block, including argument strings, environment strings, auxiliary information . . . (size varies)	
	Unspecified	
	Null auxiliary vector entry	
	Auxiliary vector . . . (2-word entries)	
	0 word	
	Environment pointers . . . (one word each)	
	0 word	
	Argument pointers . . .	
`4(%esp)`	(*Argument count* words)	
`0(%esp)`	Argument count	
	Undefined	*Low addresses*

Argument strings, environment strings, and the auxiliary information appear in no specific order within the information block; the system makes no guarantees about their arrangement. The system also may leave an unspecified amount of memory between the null auxiliary vector entry and the beginning of the information block.

General and floating-point register values are unspecified at process entry, with the exceptions appearing below. Consequently, a program that requires registers to have specific values must set them explicitly during process initialization. It should *not* rely on the operating system to set all registers to 0.

`%ebp` The content of this register is unspecified at process initialization time, but the user code should mark the deepest stack frame by setting the frame pointer to zero. No other frame's `%ebp` should have a zero value.

`%esp` Performing its usual job, the stack pointer holds the address of the bottom of the stack, which is guaranteed to be word aligned.

`%edx` In a conforming program, this register contains a function pointer that the application should register with `atexit`(BA_OS). This function is used for shared object termination code [see ''Dynamic Linking'' in Chapter 5 of the *System V ABI*].

`%cs`, `%ds`, `%es`, `%ss`
The segment registers are initialized so that the user process can address the code, data, and stack segments using a 32-bit virtual address. A program that alters their values does not conform to the ABI and has undefined behavior.

Every process has a stack, but the system defines *no* fixed stack address. Furthermore, a program's stack address can change from one system to another—even from one process invocation to another. Thus the process initialization code must use the stack address in `%esp`. Data in the stack segment at addresses below the stack pointer contain undefined values.

Whereas the argument and environment vectors transmit information from one application program to another, the auxiliary vector conveys information from the operating system to the program. This vector is an array of the following structures, interpreted according to the `a_type` member.

Figure 3-32: Auxiliary Vector

```
typedef struct
{
      int    a_type;
      union {
            long   a_val;
            void   *a_ptr;
            void   (*a_fcn)();
      } a_un;
} auxv_t;
```

Figure 3-33: Auxiliary Vector Types, `a_type`

Name	Value	a_un	
AT_NULL	0	ignored	
AT_IGNORE	1	ignored	
AT_EXECFD	2	a_val	
AT_PHDR	3	a_ptr	
AT_PHENT	4	a_val	
AT_PHNUM	5	a_val	
AT_PAGESZ	6	a_val	
AT_BASE	7	a_ptr	
AT_FLAGS	8	a_val	
AT_ENTRY	9	a_ptr	
AT_LIBPATH	10	a_val	M
AT_FPHW	11	a_val	M
AT_INTP_DEVICE	12	a_val	M
AT_INTP_INODE	13	a_val	M

AT_NULL — The auxiliary vector has no fixed length; instead its last entry's `a_type` member has this value.

AT_IGNORE This type indicates the entry has no meaning. The corresponding value of a_un is undefined.

AT_EXECFD As Chapter 5 describes, exec(BA_OS) may pass control to an interpreter program. When this happens, the system places either an entry of type AT_EXECFD or one of type AT_PHDR in the auxiliary vector. The entry for type AT_EXECFD uses the a_val member to contain a file descriptor open to read the application program's object file.

AT_PHDR Under some conditions, the system creates the memory image of the application program before passing control to the interpreter program. When this happens, the a_ptr member of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image. If the AT_PHDR entry is present, entries of types AT_PHENT, AT_PHNUM, and AT_ENTRY must also be present. See Chapter 5 in both the System V ABI and the processor supplement for more information about the program header table.

AT_PHENT The a_val member of this entry holds the size, in bytes, of one entry in the program header table to which the AT_PHDR entry points.

AT_PHNUM The a_val member of this entry holds the number of entries in the program header table to which the AT_PHDR entry points.

AT_PAGESZ If present, this entry's a_val member gives the system page size, in bytes. The same information also is available through sysconf(BA_OS).

AT_BASE The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See ''Program Header'' in the System V ABI for more information about the base address.

AT_FLAGS If present, the a_val member of this entry holds one-bit flags. Bits with undefined semantics are set to zero.

AT_ENTRY The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.

AT_LIBPATH The a_val member of this entry is non-zero if the dynamic linker should examine LD_LIBRARY_PATH when searching for shared objects of the process based on the security considerations in the Shared Object Dependency section in Chapter 5 of the gABI.

AT_FPHW The a_val member of this entry will be set to

Figure 3-34: AT_FPHW values

Value	Meaning
0	if no floating point support exists
1	if floating point software emulation exists
2	if it has a 80287 chip
3	if it has a 80387 chip or a 80487 chip

AT_INTP_DEVICE
The a_val member of this entry holds the device number of the file from which the dynamic linker is loaded.

AT_INTP_INODE
The a_val member of this entry holds the inode of the file from which the dynamic linker is loaded.

Other auxiliary vector types are reserved. No flags are currently defined for AT_FLAGS, on the Intel386 architecture.

To illustrate, suppose an example process receives two arguments.

- echo
- abi

It also inherits two environment strings (this example is not intended to show a fully configured execution environment).

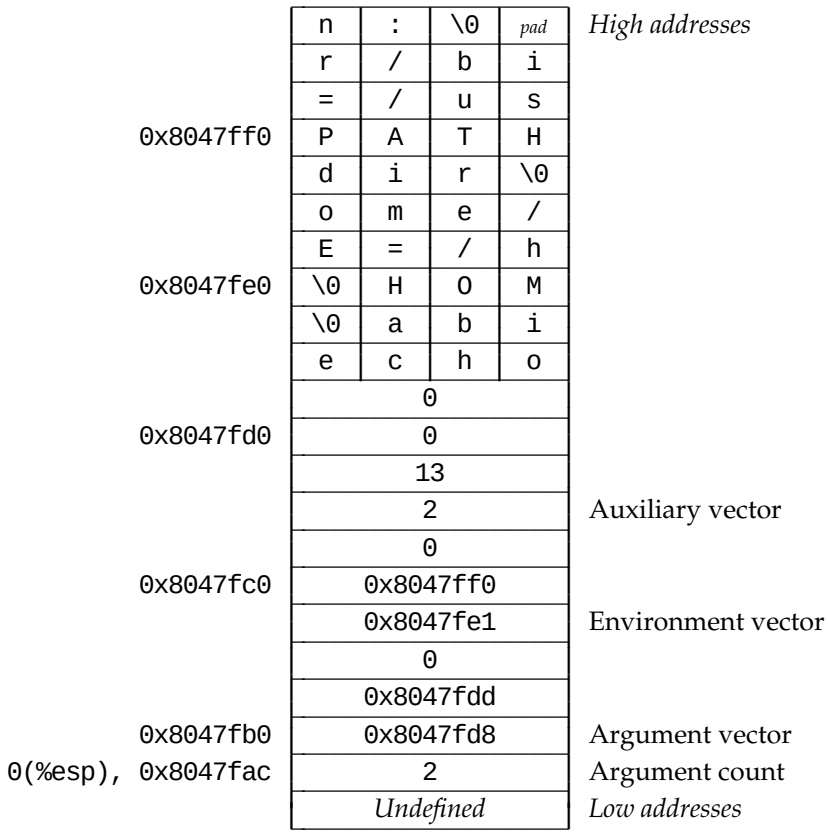
- HOME=/home/dir
- PATH=/usr/bin:

Its one non-null auxiliary vector entry holds a file descriptor.

- {AT_EXECFD, 13}

The resulting stack resides below 0x8048000, growing toward lower addresses.

Figure 3-35: Example Process Stack



Coding Examples

This section discusses example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections discuss how a program may use the machine or the operating system, and they specify what a program may and may not assume about the execution environment. Unlike previous material, the information here illustrates how operations *may* be done, not how they *must* be done.

As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does *not* prevent a program from conforming to the ABI. Two main object code models are available.

- *Absolute code*. Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses coincide with the process's virtual addresses.
- *Position-independent code*. Instructions under this model hold relative addresses, *not* absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

Following sections describe the differences between these models. Code sequences for the models (when different) appear together, allowing easier comparison.

NOTE

Examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences nor to reproduce compiler output.

NOTE

When other sections of this document show assembly language code sequences, they typically show only the absolute versions. Information in this section explains how position-independent code would alter the examples.

Code Model Overview

When the system creates a process image, the executable file portion of the process has fixed addresses, and the system chooses shared object library virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared objects conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without having to change the segment images. Thus multiple processes can share a single shared object text segment, even though the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques.

- Control transfer instructions hold offsets relative to the extended instruction pointer (EIP). An EIP-relative branch or function call computes its destination address in terms of the current instruction pointer, *not* relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in the instructions, the compiler generates code to calculate an absolute address during execution.

Because the Intel386 architecture provides EIP-relative call and branch instructions, compilers can satisfy the first condition easily.

A *global offset table* provides information for address calculation. Position-independent object files (executable and shared object files) have this table in their data segment. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual addresses as assigned for an individual process. Because data segments are private for each process, the table entries can change—unlike text segments, which multiple processes share.

Assembly language examples below show the explicit notation needed for position-independent code.

name`@GOT(%ebx)`
: This expression denotes an `%ebx`-relative reference to the global offset table entry for the symbol *name*. The `%ebx` register contains the absolute address of the global offset table, as explained below.

name`@GOTOFF(%ebx)`
: This expression denotes an `%ebx`-relative reference to the symbol *name*. Again, `%ebx` holds the global offset table address. Note this expression references *name*, not the global offset table entry for *name*.

name@PLT This expression denotes an EIP-relative reference to the procedure linkage table entry for the symbol *name*.

_GLOBAL_OFFSET_TABLE_
The symbol _GLOBAL_OFFSET_TABLE_ is used to access the global offset table. When an instruction uses the symbol, it sees the offset between the current instruction and the global offset table as the symbol value.

Position-Independent Function Prologue

This section describes the function prologue for position-independent code. A function's prologue allocates the local stack space, saves any registers it must preserve, and sets register %ebx to the global offset table's address. Because %ebx is private for each function and preserved across function calls, a function calculates its value once at the entry.

Figure 3-36: Calculating Global Offset Table Address

Line	Code
1	`call .L1`
2	`.L1: popl %ebx`
3	`addl $_GLOBAL_OFFSET_TABLE_+[.-.L1], %ebx`

These three lines accomplish the following.

1. The call instruction pushes the *absolute* address of the next instruction onto the stack.
2. Consequently, the popl instruction pops the absolute address of .L1 into register %ebx.
3. The last instruction computes the desired absolute value into %ebx. This works because _GLOBAL_OFFSET_TABLE_ in the expression gives the distance from the addl instruction to the global offset table; [.-.L1] gives the distance from .L1 to the addl instruction. Adding their sum to the absolute address of .L1, already in %ebx, gives the absolute address of the global offset table.

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

This computation can be added to the standard function prologue, giving the standard prologue for position-independent code. To illustrate, the following function prologue allocates 80 bytes of local stack space and saves the local registers `%ebx`, `%esi`, and `%edi`.

Figure 3-37: Position-Independent Function Prologue

```
prologue:
        pushl  %ebp
        movl   %esp, %ebp
        subl   $80, %esp
        pushl  %edi
        pushl  %esi
        pushl  %ebx
        call   .L1
.L1:    popl   %ebx
        addl   $_GLOBAL_OFFSET_TABLE_+[.-.L1], %ebx
```

Position-independent and absolute code use the same function epilogue.

Data Objects

This discussion excludes stack-resident objects, because programs always compute their virtual addresses relative to the stack and frame pointers. Instead, this section describes objects with static storage duration.

In the Intel386 architecture, all memory reference instructions can address any location within the 32-bit address space. Symbolic references in absolute code put the symbols' values—or absolute virtual addresses—into instructions.

Figure 3-38: Absolute Data Access

C

```
extern int src;
extern int dst;
extern int *ptr;
ptr = &dst;

*ptr = src;
```

Assembly

```
.globl   src, dst, ptr


movl     $dst, ptr

movl     ptr, %eax
movl     src, %edx
movl     %edx, (%eax)
```

Position-independent instructions cannot contain absolute addresses. Instead, instructions that reference symbols hold the symbols' offsets into the global offset table. Combining the offset with the global offset table address in `%ebx` gives the absolute address of the table entry holding the desired address.

Figure 3-39: Position-Independent Data Access

C

```
extern int src;
extern int dst;
extern int *ptr;
ptr = &dst;



*ptr = src;
```

Assembly

```
.globl   src, dst, ptr


movl     ptr@GOT(%ebx), %eax
movl     dst@GOT(%ebx), %edx
movl     %edx, (%eax)

movl     ptr@GOT(%ebx), %eax
movl     (%eax), %eax
movl     src@GOT(%ebx), %edx
movl     (%edx), %edx
movl     %edx, (%eax)
```

Finally, position-independent references to static data may be optimized. Because `%ebx` holds a known address, the global offset table, a program may use it as a base register. External references should use the global offset table entry, because dynamic linking may bind the entry to a definition outside the current object file's scope.

Figure 3-40: Position-Independent Static Data Access

C	Assembly
`static int src;`	
`static int dst;`	
`static int *ptr;`	
`ptr = &dst;`	`leal ptr@GOTOFF(%ebx), %eax`
	`leal dst@GOTOFF(%ebx), %edx`
	`movl %edx, (%eax)`
`*ptr = src;`	`movl ptr@GOTOFF(%ebx), %eax`
	`movl src@GOTOFF(%ebx), %edx`
	`movl %edx, (%eax)`

Function Calls

Programs use the `call` instruction to make direct function calls. A `call` instruction's destination is an EIP-relative value that can reach any address in the 32-bit virtual space. Even when the code for a function resides in a shared object, the caller uses the same assembly language instruction sequence, although in that case control passes from the original call, through an indirection sequence, to the desired destination. See ''Procedure Linkage Table'' in Chapter 5 for more information on the indirection sequence.

Figure 3-41: Absolute Direct Function Call

C	Assembly
`extern void function();`	`.globl function`
`function();`	`call function`

Dynamic linking may redirect a function call outside the current object file's scope; so position-independent calls should use the procedure linkage table explicitly.

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

Figure 3-42: Position-Independent Direct Function Call

C

```
extern void function();
function();
```

Assembly

```
.globl   function
call     function@PLT
```

Indirect function calls use the indirect `call` instruction.

Figure 3-43: Absolute Indirect Function Call

C

```
extern void (*ptr)();
extern void name();
ptr = name;

(*ptr)();
```

Assembly

```
.globl   ptr, name;

movl     $name, ptr

call     *ptr
```

For position-independent code, the global offset table supplies absolute addresses for all required symbols, whether the symbols name objects or functions.

Figure 3-44: Position-Independent Indirect Function Call

C

```
extern void (*ptr)();
extern void name();
ptr = name;



(*ptr)();
```

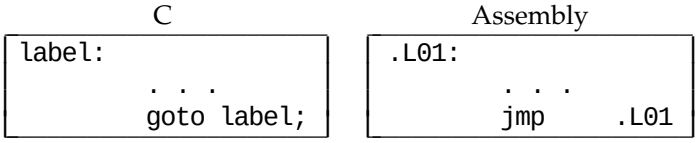
Assembly

```
.globl   ptr, name

movl     ptr@GOT(%ebx), %eax
movl     name@GOT(%ebx), %edx
movl     %edx, (%eax)

movl     ptr@GOT(%ebx), %eax
call     *(%eax)
```

Branching

Programs use branch instructions to control their execution flow. As defined by the Intel386 architecture, branch instructions hold an EIP-relative value with a signed 32-bit range, allowing a jump to any location within the virtual address space.

Figure 3-45: Branch Instruction, All Models



C `switch` statements provide multiway selection. When the `case` labels of a `switch` statement satisfy grouping constraints, the compiler implements the selection with an address table. The following examples use several simplifying conventions to hide irrelevant details:

- The selection expression resides in register `%eax;`
- `case` label constants begin at zero;
- `case` labels, `default`, and the address table use assembly names `.Lcase`*i*, `.Ldef`, and `.Ltab`, respectively.

Address table entries for absolute code contain virtual addresses; the selection code extracts an entry's value and jumps to that address. Position-independent table entries hold offsets; the selection code computes a destination's absolute address.

Figure 3-46: Absolute `switch` Code

C

```
switch (j)
{
case 0:
     . . .
case 2:
     . . .
case 3:
     . . .
default:
     . . .
}
```

Assembly

```
          cmpl    $3, %eax
          ja      .Ldef
          jmp     *.Ltab(,%eax,4)
.Ltab:    .long   .Lcase0
          .long   .Ldef
          .long   .Lcase2
          .long   .Lcase3
```

Figure 3-47: Position-Independent `switch` Code

C

```
switch (j)
{
case 0:
     . . .
case 2:
     . . .
case 3:
     . . .
default:
     . . .
}
```

Assembly

```
     cmpl    $3, %eax
     ja      .Ldef
     leal    .Ltab@GOTOFF(%ebx), %edx
     movl    (%edx,%eax,4), %eax
     movl    .Ltab@GOTOFF(%ebx,%eax,4), %eax
     call    .Ljmp
.Ljmp:
     popl    %ecx
     addl    %ecx, %eax
     jmp     *%eax
.Ltab:
     .long   .Lcase0 - .Ljmp
     .long   .Ldef - .Ljmp
     .long   .Lcase2 - .Ljmp
     .long   .Lcase3 - .Ljmp
```

C Stack Frame

Figure 3-48 shows the C stack frame organization. It conforms to the standard stack frame with designated roles for unspecified areas in the standard frame. M
This represents one possible organization of the C stack frame. Usage of `%ebp` as a M
frame pointer, the exact positions of the callee saved registers, and space for local M
storage is implementation specific.

Figure 3-48: C Stack Frame

Base	Offset	Contents	
`%ebp`	4*n*+8	argument word *n*	*High addresses*
		. . .	
	8	argument word 0	
	4	return address	
`%ebp`	0	caller's `%ebp`	
`%ebp`	–4	*x* words local space: automatic variables, temporaries,	
`%ebp`	–4*x*	etc.	
`%esp`	12		
`%esp`	8	caller's `%edi`	
	4	caller's `%esi`	
`%esp`	0	caller's `%ebx`	*Low addresses*

A C stack frame doesn't normally change size during execution. The exception is dynamically allocated stack memory, discussed below. By convention, a function allocates automatic (local) variables in the middle of its frame and references them as negative offsets from `%ebp`. Its incoming arguments reside in the previous frame, referenced as positive offsets from `%ebp`. If necessary, a function saves the values of `%edi`, `%esi`, and `%ebx` in the positions shown and restores their values before returning to the caller. The positions may be different from the diagram above, depending on which of these three registers the function saves and restores.

Variable Argument List

Previous sections describe the rules for passing arguments. Unfortunately, some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments reside on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many machines, including the Intel386. Nonetheless, portable C programs should use the facilities defined in the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists.

Allocating Stack Space Dynamically

Unlike some other languages, C does not need dynamic stack allocation *within* a stack frame. Frames are allocated dynamically on the program stack, depending on program execution, but individual stack frames can have static sizes. Nonetheless, the architecture supports dynamic allocation for those languages that require it, and the standard calling sequence and stack frame support it as well. Thus languages that need dynamic stack frame sizes can call C functions, and vice versa.

Figure 3-48 shows the layout of the C stack frame. The double line divides the area referenced from `%ebp` from the area referenced from `%esp`. Dynamic space is allocated below the line, as a downward growing heap whose size changes as required. Typical C functions have no space in the heap. All areas above the heap in the current frame have a known size to the compiler. Dynamic stack allocation thus takes the following steps.

1. Stack frames are word aligned; dynamic allocation should preserve this property. Thus the program rounds (up) the desired byte count to a multiple of 4.
2. The program decreases the stack pointer by the rounded byte count, increasing its frame size. At this point, the ''new'' space resides just below the register save area at the bottom of the stack.
3. The program copies the register save area (three or fewer words) to the bottom of the stack, effectively moving the new space up into the frame.

NOTE

The register save area is reserved and should not be used for purposes outside of this document. G

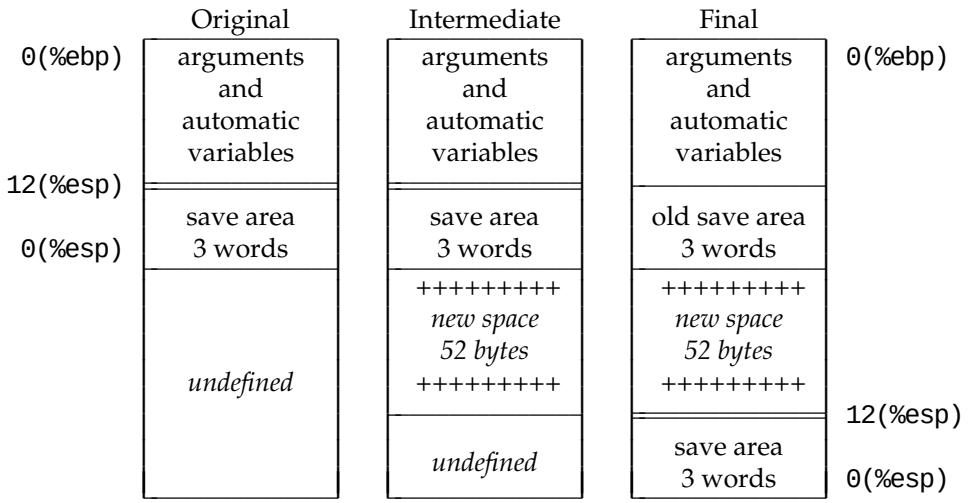
Even in the presence of signals, dynamic allocation is ''safe.'' If a signal interrupts allocation, one of three things can happen.

- The signal handler can return. The process then resumes the dynamic allocation from the point of interruption.
- The signal handler can execute a non-local goto, or `longjmp` [see `setjmp`(BA_LIB)]. This resets the process to a new context in a previous stack frame, automatically discarding the dynamic allocation.
- The process can terminate.

Regardless of when the signal arrives during dynamic allocation, the result is a consistent (though possibly dead) process.

To illustrate, assume a program wants to allocate 50 bytes, and it has saved three registers in the bottom of the frame. The first step is rounding 50 to 52, making it a multiple of 4. Figure 3-49 shows how the stack frame changes.

Figure 3-49: Dynamic Stack Allocation



New space starts at `12(%esp)`. As described, every dynamic allocation in *this* function will return a new area starting at `12(%esp)`, leaving previous heap objects untouched (other functions could have different heap addresses).

Consequently, the compiler should compute the absolute address for each area, avoiding relative references. Otherwise, future allocations in the same frame would destroy the heap's integrity.

Existing stack objects reside at fixed offsets from the frame pointer (`%ebp`). Dynamic allocation preserves those offsets, because the frame pointer does not change and the objects relative to it do not move. Objects relative to the stack pointer (`%esp`) move, but their `%esp`-relative positions do not change. Accordingly, compilers arrange not to publicize the absolute address of any object in the bottom half of the stack frame (in a way that violates the scope rules). `%esp`-relative references stay valid after dynamic allocation, but absolute addresses do not.

No special code is needed to free dynamically allocated stack memory. The function return resets the stack pointer and removes the entire stack frame, including the heap, from the stack. Naturally, a program should not reference heap objects after they have gone out of scope.

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

4 OBJECT FILES

ELF Header 4-1
Machine Information 4-1

Sections 4-2
Special Sections 4-2

Symbol Table 4-3
Symbol Values 4-3

Relocation 4-4
Relocation Types 4-4

ELF Header

Machine Information

For file identification in `e_ident`, the Intel386 architecture requires the following values.

Figure 4-1: Intel386 Identification, `e_ident`

Position	Value
`e_ident[EI_CLASS]`	`ELFCLASS32`
`e_ident[EI_DATA]`	`ELFDATA2LSB`

Processor identification resides in the ELF header's `e_machine` member and must have the value `EM_386`.

The ELF header's `e_flags` member holds bit flags associated with the file. The Intel386 architecture defines no flags; so this member contains zero.

Sections

Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

Figure 4-2: Special Sections

Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

`.got` This section holds the global offset table. See ''Coding Examples'' in Chapter 3 and ''Global Offset Table'' in Chapter 5 for more information.

`.plt` This section holds the procedure linkage table. See ''Procedure Linkage Table'' in Chapter 5 for more information.

DRAFT COPY
March 19, 1997
File: abi_386/chap4 (Delta 44.3)
386:adm.book:sum

Symbol Table

Symbol Values

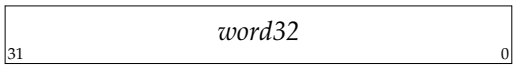
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This signals to the dynamic linker that the symbol definition for that function is not contained in the executable file itself. If that symbol has been allocated a procedure linkage table entry in the executable file, and the `st_value` member for that symbol table entry is non-zero, the value will contain the virtual address of the first instruction of that procedure linkage table entry. Otherwise, the `st_value` member contains zero. This procedure linkage table entry address is used by the dynamic linker in resolving references to the address of the function. See ''Function Addresses'' in Chapter 5 for details.

Relocation

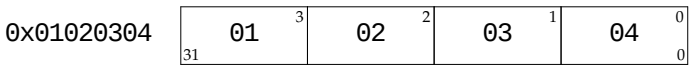
Relocation Types

Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners).

Figure 4-3: Relocatable Fields



word32 This specifies a 32-bit field occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the Intel386 architecture.



Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It first decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

A This means the addend used to compute the value of the relocatable field.

B This means the base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different. See ''Program Header'' in the System V ABI for more information about the base address.

G This means the offset into the global offset table at which the address of the relocation entry's symbol will reside during execution. See ''Coding Examples'' in Chapter 3 and ''Global Offset Table'' in Chapter 5 for more information.

DRAFT COPY
March 19, 1997
File: abi_386/chap4 (Delta 44.3)
386:adm.book:sum

GOT This means the address of the global offset table. See ''Coding Examples'' in Chapter 3 and ''Global Offset Table'' in Chapter 5 for more information.

L This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See ''Procedure Linkage Table'' in Chapter 5 for more information.

P This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

S This means the value of the symbol whose index resides in the relocation entry.

A relocation entry's `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. The Intel386 architecture uses only `Elf32_Rel` relocation entries, the field to be relocated holds the addend. In all cases, the addend and the computed result use the same byte order.

Figure 4-4: Relocation Types

Name	Value	Field	Calculation
R_386_NONE	0	none	none
R_386_32	1	*word32*	S + A
R_386_PC32	2	*word32*	S + A - P
R_386_GOT32	3	*word32*	G + A - P
R_386_PLT32	4	*word32*	L + A - P
R_386_COPY	5	none	none
R_386_GLOB_DAT	6	*word32*	S
R_386_JMP_SLOT	7	*word32*	S
R_386_RELATIVE	8	*word32*	B + A
R_386_GOTOFF	9	*word32*	S + A - GOT
R_386_GOTPC	10	*word32*	GOT + A - P

Some relocation types have semantics beyond simple calculation.

R_386_GOT32 This relocation type computes the distance from the base of the global offset table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.

R_386_PLT32 This relocation type computes the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_386_COPY The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

R_386_GLOB_DAT This relocation type is used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.

R_386_JMP_SLOT The link editor creates this relocation type for dynamic linking. Its offset member gives the location of a procedure linkage table entry. The dynamic linker modifies the procedure linkage table entry to transfer control to the designated symbol's address [see ''Procedure Linkage Table'' in Chapter 5].

R_386_RELATIVE The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_386_GOTOFF This relocation type computes the difference between a symbol's value and the address of the global offset table. It additionally instructs the link editor to build the global offset table.

R_386_GOTPC This relocation type resembles R_386_PC32, except it uses the address of the global offset table in its calculation. The symbol referenced in this relocation normally is _GLOBAL_OFFSET_TABLE_, which additionally instructs the link editor to build the global offset table.

5 PROGRAM LOADING AND DYNAMIC LINKING

Program Loading 5-1

Dynamic Linking 5-5
Dynamic Section 5-5
Global Offset Table 5-5
Function Addresses 5-6
Procedure Linkage Table 5-7
Program Interpreter 5-10

Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When—and if—the system physically reads the file depends on the program's execution behavior, system load, and so on. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for the Intel386 architecture segments are congruent modulo 4 KB (`0x1000`) or larger powers of 2. Because 4 KB is the maximum page size, the files will be suitable for paging regardless of physical page size.

Figure 5-1: Executable File

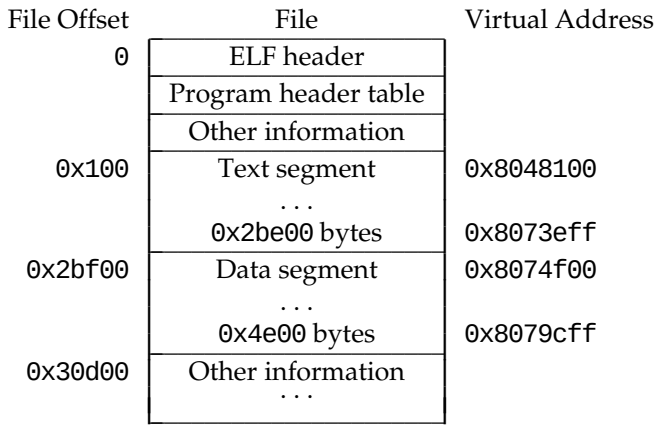


Figure 5-2: Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000

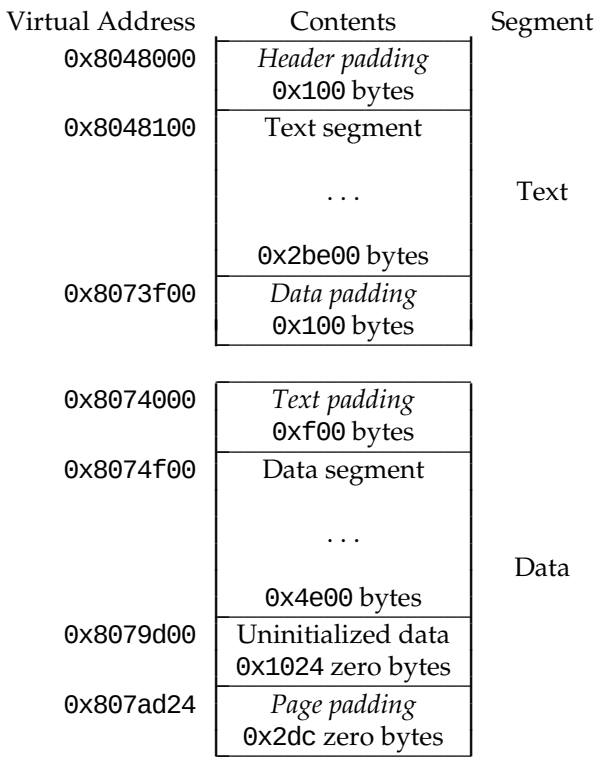
Although the example's file offsets and virtual addresses are congruent modulo 4 KB for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the example above, the region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. ''Impurities'' in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 4 KB (0x1000) pages.

Figure 5-3: Process Image Segments



One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code (see ''Coding Examples'' in Chapter 3). To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments' *relative positions*. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

DRAFT COPY
March 19, 1997
File: abi_386/chap5 (Delta 44.3)
386:adm.book:sum

Figure 5-4: Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0x80000200	0x8002a400	0x80000000
Process 2	0x80081200	0x800ab400	0x80081000
Process 3	0x900c0200	0x900ea400	0x900c0000
Process 4	0x900c6200	0x900f0400	0x900c6000

DRAFT COPY
March 19, 1997
File: abi_386/chap5 (Delta 44.3)
386:adm.book:sum

Dynamic Linking

Dynamic Section

Dynamic section entries give information to the dynamic linker. Some of this information is processor-specific, including the interpretation of some entries in the dynamic structure.

DT_PLTGOT — On the Intel386 architecture, this entry's `d_ptr` member gives the address of the first entry in the global offset table. As mentioned below, the first three global offset table entries are reserved, and two are used to hold procedure linkage table information.

Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries [see ''Relocation'' in Chapter 4]. After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which will be type `R_386_GLOB_DAT` referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image. On the Intel386 architecture, entries one and two in the global offset table also are reserved. ''Procedure Linkage Table'' below describes them.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For the Intel386 architecture, the symbol _GLOBAL_OFFSET_TABLE_ may be used to access the table.

Figure 5-5: Global Offset Table

```
extern Elf32_Addr   _GLOBAL_OFFSET_TABLE_[];
```

The symbol _GLOBAL_OFFSET_TABLE_ may reside in the middle of the .got section, allowing both negative and non-negative ''subscripts'' into the array of addresses.

Function Addresses

References to the address of a function from an executable file and the shared objects associated with it might not resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined in a shared object will normally be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. [See ''Symbol Values'' in Chapter 4]. The dynamic linker

treats such symbol table entries specially. If the dynamic linker is searching for a symbol, and encounters a symbol table entry for that symbol in the executable file, it normally follows the rules below.

1. If the `st_shndx` member of the symbol table entry is not `SHN_UNDEF`, the dynamic linker has found a definition for the symbol and uses its `st_value` member as the symbol's address.
2. If the `st_shndx` member is `SHN_UNDEF` and the symbol is of type `STT_FUNC` and the `st_value` member is not zero, the dynamic linker recognizes this entry as special and uses the `st_value` member as the symbol's address.
3. Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the Intel386 architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

Figure 5-6: Absolute Procedure Linkage Table

```
.PLT0: pushl  got_plus_4
       jmp    *got_plus_8
       nop; nop
       nop; nop
.PLT1: jmp    *name1_in_GOT
       pushl  $offset
       jmp    .PLT0@PC
.PLT2: jmp    *name2_in_GOT
       pushl  $offset
       jmp    .PLT0@PC
       ...
```

Figure 5-7: Position-Independent Procedure Linkage Table

```
.PLT0: pushl  4(%ebx)
       jmp    *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp    *name1@GOT(%ebx)
       pushl  $offset
       jmp    .PLT0@PC
.PLT2: jmp    *name2@GOT(%ebx)
       pushl  $offset
       jmp    .PLT0@PC
       ...
```

NOTE

As the figures show, the procedure linkage table instructions use different operand addressing modes for absolute code and for position-independent code. Nonetheless, their interfaces to the dynamic linker are the same.

Following the steps below, the dynamic linker and the program ''cooperate'' to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. If the procedure linkage table is position-independent, the address of the global offset table must reside in `%ebx`. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. Consequently, the calling function is responsible for setting the global offset table base register before calling the procedure linkage table entry.
3. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.
4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially, the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.
5. Consequently, the program pushes a relocation offset (*offset*) on the stack. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type `R_386_JMP_SLOT`, and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, `name1` in this case.
6. After pushing the relocation offset, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushl` instruction places the value of the second global offset table entry (*got_plus_4* or `4(%ebx)`) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (*got_plus_8* or `8(%ebx)`), which transfers control to the dynamic linker.
7. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the ''real'' address for `name1` in its global offset table entry, and transfers control to the desired destination.

8. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of ''falling through'' to the `pushl` instruction.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_386_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

NOTE

Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

Program Interpreter G

There is one valid program interpreter for programs conforming to the Intel386 ABI: G

```
/usr/lib/libc.so.1                                    G
```

6 LIBRARIES

Shared Library Names 6-1

C Library 6-2
Additional Entry Points 6-2
Support Routines 6-3
Global Data Symbols 6-4
- Application Constraints 6-4

System Data Interfaces 6-5
Data Definitions 6-5
- Reentrancy Considerations 6-5

X Window Data Definitions 6-127
Motif 1.2 Data Definitions 6-192
TCP/IP Data Definitions 6-269

Shared Library Names

The version number of the libraries named in the *System V Generic ABI* is specified below.

Figure 6-1: Shared Library Names

Library Reference Name
`libc.so.1`
`libthread.so.1`
`libdl.so.1`
`libnsl.so.1`
`libX11.so.5.0`
`libXt.so.5.0`
`libXext.so.5.0`
`libXm.so.1.2`
`libMrm.so.1.2`

C Library M

Additional Entry Points

The following routines are included in the **libc** library to provide entry points for the required source-level interface listed in the *System V ABI*. A description and syntax symmary for each function follows the table. M

Figure 6-2: `libc` **Additional Required Entry Points**

`_fxstat _lxstat _xmknod _xstat nuname _nuname`

`int _fxstat(int, int, struct stat *);`
The semantics of this function are identical to those of the `fstat`(BA_OS) function described in the *System V Interface Definition, Edition 4*. Its only difference is that it requires an extra first argument whose value must be 2.

`int _lxstat(int, char *, struct stat *);`
The semantics of this function are identical to those of the `lstat`(BA_OS) function described in the *System V Interface Definition, Edition 4*. Its only difference is that it requires an extra first argument whose value must be 2.

`int nuname(struct utsname *);`
The semantics and syntax of this function are identical to those of the `uname`(BA_OS) function described in the *System V Interface Definition, Edition 4*. The symbol `_nuname` is also available with the same semantics.

`int _xmknod(int, char *, mode_t, dev_t);`
The semantics of this function are identical to those of the `mknod`(BA_OS) function described in the *System V Interface Definition, Edition 4*. Its only difference is that it requires an extra first argument whose value must be 2.

`int _xstat(int, char *, struct stat *);`
The semantics of this function are identical to those of the `stat`(BA_OS) function described in the *System V Interface*

Definition, Edition 4. Its only difference is that it requires an extra first argument whose value must be 2.

Support Routines

Besides operating system services, **libc** contains the following processor-specific support routines. M

Figure 6-3: `libc`**, Support Routines**

`_fpstart`	`_ _fpstart`	`sbrk`	`_sbrk`	

`char *sbrk(int incr);`

This function adds *incr* bytes to the *break value* and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased. The break value is the address of the first allocation beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process, its contents are undefined. Upon successful completion, `sbrk` returns the old break value. Otherwise, it returns `-1` and sets `errno` to indicate the error. The symbol `_sbrk` is also available with the same semantics.

`void __fpstart(void);`

This function calls `_fpstart()`, to initialize the floating-point environment.

`void _fpstart(void);`

This function initializes the floating-point execution environment. It sets _fp_hw to the appropriate value. It sets the rounding mode to ''nearest.'' It also resets the Intel387 control word to the default state.

Global Data Symbols

The libc library requires that some globel external data objects be defined for the routines to work properly. In addition to the corresponding data symbols listed in the *System V ABI*, the following symbols must be provided in the system library on all ABI-conforming systems implemented with the Intel386 architecture. Declarations for the data objects listed below can be found in the Data Definitions section of this chapter or immediately following the table. M

Figure 6-4: libc, Global External Data Symbols

__flt_rounds	_fp_hw	__huge_val

```
extern int _fp_hw;
```

This variable describes the floating-point hardware available. If the value is zero, no floating-point support is present. If the value is 1, the floating-point support is provided by an Intel387 software emulator. If the value is 2, an 80287 chip is available. If the value is 3, an Intel387 chip is available. System software sets the value appropriately, before transferring control to main.

Application Constraints

As described above, libc provides symbols for applications. In a few cases, however, an application is obliged to provide symbols for the library. In addition to the application-provided symbols listed in this section of the *System V ABI*, conforming applications on the Intel386 architecture are also required to provide the following symbols. M

```
extern _end;
```

This symbol refers neither to a routine nor to a location with interesting contents. Instead, its address must correspond to the beginning of a program's dynamic allocation area, called the heap. Typically, the heap begins immediately after the data segment of the program's executable file.

```
extern const int _lib_version;
```

This variable's value specifies the compilation and execution mode for the program. If the value is zero, the program wants to preserve the semantics of older (pre-ANSI) C, where conflicts exist with ANSI. Otherwise, the value is non-zero, and the program wants ANSI C semantics.

DRAFT COPY
March 19, 1997
File: abi_386/chap6 (Delta 44.21)
386:adm.book:sum

System Data Interfaces

Data Definitions

This section contains standard data definitions that describe system data. These files are referred to by their names in angle brackets: <*name.h*> and <sys/*name.h*>. Included in these data definitions are macro definitions and data definitions.

The data objects described in this section are part of the interface between an ABI-conforming application and the underlying ABI-conforming system where it will run. While an ABI-conforming system must provide these interfaces, it is not required to contain the actual data definitions referenced here. Programmers should observe that the sources of the structures defined in these data definitions are defined in SVID.

ANSI C serves as the ABI reference programming language, and data definitions are specificed in ANSI C format. The C language is used here as a convenient notation. Using a C language description of these data objects does *not* preclude their use by other programming languages. M

Reentrancy Considerations M

New conventions have been added to accomodate the new requirements of reentrancy. Some historic binary code sequences are inherently non-reentrant. Unless great care is taken, multi-threaded applications cannot safely use such sequences. The most portable (i.e. those guaranteed to work in all cases) are those that are marked as reentrant in this chapter. For the ABI, this sometimes requires that two definitions exist for these interfaces, one that is reentrant and one that is not. These are indicated by comments that define which of the alternate definitions is reentrant. These alternatives are not selected at run-time, but are intended to be bound at application build time. M M M M M M M M

NOTE

All information presented in the figures marked with * are new to the Fourth Edition of the psABI. M

Figure 6-5: <aio.h>*

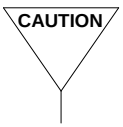
```
struct  aiocb {
       int             aio_fildes;
       volatile void*  aio_buf;
       size_t          aio_nbytes;
       off_t           aio_offset;
       int             aio_reqprio;
       struct sigevent aio_sigevent;
       int             aio_lio_opcode;
       ssize_t         ;
       int             ;
       int             ;
       void            ;
       int             ;
} ;

#define AIO_CANCELED    (0)
#define AIO_ALLDONE     (1)
#define AIO_NOTCANCELED (2)

#define LIO_NOWAIT      (0)
#define LIO_WAIT        (1)
#define LIO_NOP         (0)
#define LIO_READ        (1)
#define LIO_WRITE       (2)
```

Figure 6-6: <assert.h>

```
extern void __assert(const char *, const char *, int);
#define assert(EX) (void)((EX)||(__assert(#EX, __FILE__, __LINE__), 0))
```

Figure 6-7: <ctype.h>

```
#define _U   01
#define _L   02
#define _N   04
#define _S   010
#define _P   020
#define _C   040
#define _B   0100
#define _X   0200

extern unsigned char      __ctype[];

#define isalpha(c)  ((__ctype+1)[c]&(_U|_L))
#define isupper(c)  ((__ctype+1)[c]&_U)
#define islower(c)  ((__ctype+1)[c]&_L)
#define isdigit(c)  ((__ctype+1)[c]&_N)
#define isxdigit(c) ((__ctype+1)[c]&_X)
#define isalnum(c)  ((__ctype+1)[c]&(_U|_L|_N))
#define isspace(c)  ((__ctype+1)[c]&_S)
#define ispunct(c)  ((__ctype+1)[c]&_P)
#define isprint(c)  ((__ctype+1)[c]&(_P|_U|_L|_N|_B))
#define isgraph(c)  ((__ctype+1)[c]&(_P|_U|_L|_N))
#define iscntrl(c)  ((__ctype+1)[c]&_C)
#define isascii(c)  (!((c)&~0177))
#define _toupper(c) ((__ctype+258)[c])
#define _tolower(c) ((__ctype+258)[c])
#define toascii(c)  ((c)&0177)
```



The data definitions in ctype.h are moved to Level 2 as of January 1, 1993. In order to correctly function in an internationalized environment, applications are encouraged to use the functions in libc instead. M

Figure 6-8: <dirent.h>

```
typedef struct {
      int          dd_fd;
      int          dd_loc;
      int          dd_size;
      char         *dd_buf;
} DIR;

struct dirent {
      ino_t        d_ino;
      off_t        d_off;
      unsigned short     d_reclen;
      char         d_name[1];
};
```

Figure 6-9: <dlfcn.h>*

```
#define RTLD_LAZY       1
#define RTLD_NOW        2
#define RTLD_GLOBAL     4
```

Figure 6-10: <elf.h>*, Part 1 of 6

```
#define ELF32_FSZ_ADDR    4
#define ELF32_FSZ_HALF    2
#define ELF32_FSZ_OFF     4
#define ELF32_FSZ_SWORD   4
#define ELF32_FSZ_WORD    4

#define EI_NIDENT   16

typedef struct {
      unsigned chare_ident[EI_NIDENT];
      Elf32_Half   e_type;
      Elf32_Half   e_machine;
      Elf32_Word   e_version;
      Elf32_Addr   e_entry;
      Elf32_Off    e_phoff;
      Elf32_Off    e_shoff;
      Elf32_Word   e_flags;
      Elf32_Half   e_ehsize;
      Elf32_Half   e_phentsize;
      Elf32_Half   e_phnum;
      Elf32_Half   e_shentsize;
      Elf32_Half   e_shnum;
      Elf32_Half   e_shstrndx;
} Elf32_Ehdr;

#define ELFMAG0           0x7f
#define ELFMAG1           'E'
#define ELFMAG2           'L'
#define ELFMAG3           'F'
#define ELFMAG            "\177ELF"
#define SELFMAG           4
```

Figure 6-11: <elf.h>*, Part 2 of 6

```
#define EI_MAG0          0
#define EI_MAG1          1
#define EI_MAG2          2
#define EI_MAG3          3
#define EI_CLASS         4
#define EI_DATA          5
#define EI_VERSION       6
#define EI_PAD           7

#define ELFCLASSNONE     0
#define ELFCLASS32       1
#define ELFCLASS64       2
#define ELFCLASSNUM      3
#define ELFDATANONE      0
#define ELFDATA2LSB      1
#define ELFDATA2MSB      2
#define ELFDATANUM       3

#define ET_NONE          0
#define ET_REL           1
#define ET_EXEC          2
#define ET_DYN           3
#define ET_CORE          4
#define ET_NUM           5

#define      ET_LOPROC   0xff00
#define      ET_HIPROC   0xffff

#define EM_NONE          0
#define EM_M32           1
#define EM_SPARC         2
#define EM_386           3
#define EM_68K           4
#define EM_88K           5
#define EM_486           6
#define EM_860           7
#define EM_NUM           8
```

Figure 6-12: <elf.h>*, Part 3 of 6

```
#define EV_NONE           0
#define EV_CURRENT        1
#define EV_NUM            2

typedef struct {
      Elf32_Word   p_type;
      Elf32_Off    p_offset;
      Elf32_Addr   p_vaddr;
      Elf32_Addr   p_paddr;
      Elf32_Word   p_filesz;
      Elf32_Word   p_memsz;
      Elf32_Word   p_flags;
      Elf32_Word   p_align;
} Elf32_Phdr;

#define PT_NULL           0
#define PT_LOAD           1
#define PT_DYNAMIC        2
#define PT_INTERP         3
#define PT_NOTE           4
#define PT_SHLIB          5
#define PT_PHDR           6
#define PT_NUM            7

#define PT_LOPROC   0x70000000
#define PT_HIPROC   0x7fffffff

#define PF_R        0x4
#define PF_W        0x2
#define PF_X        0x1

#define PF_MASKPROC 0xf0000000
```

Figure 6-13: <elf.h>*, Part 4 of 6

```
typedef struct {
      Elf32_Word   sh_name;
      Elf32_Word   sh_type;
      Elf32_Word   sh_flags;
      Elf32_Addr   sh_addr;
      Elf32_Off    sh_offset;
      Elf32_Word   sh_size;
      Elf32_Word   sh_link;
      Elf32_Word   sh_info;
      Elf32_Word   sh_addralign;
      Elf32_Word   sh_entsize;
} Elf32_Shdr;

#define SHT_NULL          0
#define SHT_PROGBITS      1
#define SHT_SYMTAB        2
#define SHT_STRTAB        3
#define SHT_RELA          4
#define SHT_HASH          5
#define SHT_DYNAMIC       6
#define SHT_NOTE          7
#define SHT_NOBITS        8
#define SHT_REL           9
#define SHT_SHLIB         10
#define SHT_DYNSYM        11
#define SHT_NUM           12

#define SHT_LOUSER  0x80000000
#define SHT_HIUSER  0xffffffff
#define      SHT_LOPROC   0x70000000
#define      SHT_HIPROC   0x7fffffff
#define SHF_MASKPROC      0xf0000000

#define SHF_WRITE         0x1
#define SHF_ALLOC         0x2
#define SHF_EXECINSTR     0x4
```

Figure 6-14: <elf.h>*, Part 5 of 6

```
#define SHN_UNDEF         0
#define SHN_LORESERVE     0xff00
#define SHN_ABS           0xfff1
#define SHN_COMMON        0xfff2
#define SHN_HIRESERVE     0xffff
#define SHN_LOPROC        0xff00
#define SHN_HIPROC        0xff1f

typedef struct {
      Elf32_Word          st_name;
      Elf32_Addr          st_value;
      Elf32_Word          st_size;
      unsigned char       st_info;
      unsigned char       st_other;
      Elf32_Half          st_shndx;
} Elf32_Sym;

#define STN_UNDEF   0

#define ELF32_ST_BIND(info)            ((info) >> 4)
#define ELF32_ST_TYPE(info)            ((info) & 0xf)
#define ELF32_ST_INFO(bind,type) (((bind)<<4)+((type)&0xf))

#define STB_LOCAL   0
#define STB_GLOBAL  1
#define STB_WEAK    2
#define STB_NUM     3
#define STB_LOPROC  13
#define STB_HIPROC  15
```

Figure 6-15: <elf.h>*, Part 6 of 6

```
#define STT_NOTYPE  0
#define STT_OBJECT  1
#define STT_FUNC    2
#define STT_SECTION 3
#define STT_FILE    4
#define STT_NUM     5
#define STT_LOPROC  13
#define STT_HIPROC  15

typedef struct {
      Elf32_Addr   r_offset;
      Elf32_Word   r_info;
} Elf32_Rel;

typedef struct {
      Elf32_Addr   r_offset;
      Elf32_Word   r_info;
      Elf32_Sword  r_addend;
} Elf32_Rela;

#define ELF32_R_SYM(info) ((info)>>8)
#define ELF32_R_TYPE(info)((unsigned char)(info))
#define ELF32_R_INFO(sym,type)   (((sym)<<8)+(unsigned char)(type))
```

Figure 6-16: <errno.h>**, Part 1 of 3**

```
#define EPERM           1
#define ENOENT          2
#define ESRCH           3
#define EINTR           4
#define EIO             5
#define ENXIO           6
#define E2BIG           7
#define ENOEXEC         8
#define EBADF           9
#define ECHILD          10
#define EAGAIN          11
#define ENOMEM          12
#define EACCES          13
#define EFAULT          14
#define ENOTBLK         15
#define EBUSY           16
#define EEXIST          17
#define EXDEV           18
#define ENODEV          19
#define ENOTDIR         20
#define EISDIR          21
#define EINVAL          22
#define ENFILE          23
#define EMFILE          24
#define ENOTTY          25
#define ETXTBSY         26
#define EFBIG           27
```

Figure 6-17: <errno.h>**, Part 2 of 3**

```
#define ENOSPC          28
#define ESPIPE          29
#define EROFS           30
#define EMLINK          31
#define EPIPE           32
#define EDOM            33
#define ERANGE          34
#define ENOMSG          35
#define EIDRM           36
#define ECHRNG          37
#define EL2NSYNC        38
#define EL3HLT          39
#define EL3RST          40
#define ELNRNG          41
#define EUNATCH         42
#define ENOCSI          43
#define EL2HLT          44
#define EDEADLK         45
#define ENOLCK          46
#define ENOSTR          60
#define ENODATA         61
#define ETIME           62
#define ENOSR           63
#define ENONET          64
#define ENOPKG          65
#define EREMOTE         66
#define ENOLINK         67
```

Figure 6-18: `<errno.h>`**, Part 3 of 3**

```
#define EADV              68
#define ESRMNT            69
#define ECOMM             70
#define EPROTO            71
#define EMULTIHOP         74
#define EBADMSG           77
#define ENAMETOOLONG      78
#define EOVERFLOW         79
#define ENOTUNIQ          80
#define EBADFD            81
#define EREMCHG           82
#define ENOSYS            89
#define ELOOP             90
#define ERESTART          91
#define ESTRPIPE          92
#define ENOTEMPTY         93
#define EUSERS            94
#define ECONNABORTED      130                          G
#define CONNRESET         131                          G
#define ECONNREFUSED                  146              G
#define EINPROGRESS       150                          M
#define ESTALE            151
#define ECANCELED         158                          M

/* Non-reentrant */                                    M
extern int errno;                                      M

/* Reentrant */                                        M
#define errno   (*__thr_errno())                       M
```

Figure 6-19: <fcntl.h>**, Part 1 of 2**

```
#define O_RDONLY          0
#define O_WRONLY          1
#define O_RDWR            2
#define O_NDELAY          0x04
#define O_APPEND          0x08
#define O_SYNC            0x10
#define O_NONBLOCK        0x80
#define O_CREAT           0x100
#define O_TRUNC           0x200
#define O_EXCL            0x400
#define O_NOCTTY          0x800

#define F_DUPFD           0
#define F_GETFD           1
#define F_SETFD           2
#define F_GETFL           3
#define F_SETFL           4
#define F_GETLK           14
#define F_SETLK           6
#define F_SETLKW          7
```

Figure 6-20: <fcntl.h>**, Part 2 of 2**

```
typedef struct flock {
      short  l_type;
      short  l_whence;
      off_t  l_start;
      off_t  l_len;
      long   l_sysid;
      pid_t  l_pid;
      long   pad[4];
} flock_t;

#define F_RDLCK           01
#define F_WRLCK           02
#define F_UNLCK           03

#define O_ACCMODE         3
#define FD_CLOEXEC        1
```

Figure 6-21: <float.h>**, Single-Precision**

```
extern int          __flt_rounds;
#define FLT_ROUNDS          __flt_rounds

#define FLT_RADIX           2
#define FLT_MANT_DIG        24
#define FLT_EPSILON         1.19209290E-07F
#define FLT_DIG             6
#define FLT_MIN_EXP         (-125)
#define FLT_MIN             1.17549435E-38F
#define FLT_MIN_10_EXP      (-37)
#define FLT_MAX_EXP         (+128)
#define FLT_MAX             3.40282347E+38F
#define FLT_MAX_10_EXP      (+38)
```

Figure 6-22: <float.h>**, Double-Precision**

```
#define DBL_MANT_DIG        53
#define DBL_EPSILON         2.2204460492503131E-16
#define DBL_DIG             15
#define DBL_MIN_EXP         (-1021)
#define DBL_MIN             2.2250738585072014E-308
#define DBL_MIN_10_EXP      (-307)
#define DBL_MAX_EXP         (+1024)
#define DBL_MAX             1.7976931348623157E+308
#define DBL_MAX_10_EXP      (+308)
```

Figure 6-23: <float.h>**, Extended-Precision**

```
#define LDBL_MANT_DIG     64                                    G
#define LDBL_EPSILON      1.084202172485504434e-19              G
#define LDBL_DIG          18                                    G
#define LDBL_MIN_EXP      -16381                                G
#define LDBL_MIN          3.362103143112093506e-4932            G
#define LDBL_MIN_10_EXP   -4931                                 G
#define LDBL_MAX_EXP      16384                                 G
#define LDBL_MAX          1.189731495347231765e+4932            G
#define LDBL_MAX_10_EXP   4932                                  G
```

Figure 6-24: <fmtmsg.h>**, Part 1 of 2**

```
#define MM_NULL           0L

#define MM_HARD           0x00000001L
#define MM_SOFT           0x00000002L
#define MM_FIRM           0x00000004L
#define MM_RECOVER        0x00000100L
#define MM_NRECOV         0x00000200L
#define MM_APPL           0x00000008L
#define MM_UTIL           0x00000010L
#define MM_OPSYS          0x00000020L
#define MM_PRINT          0x00000040L
#define MM_CONSOLE        0x00000080L
```

Figure 6-25: <fmtmsg.h>**, Part 2 of 2**

```
#define MM_NOSEV          0
#define MM_HALT           1
#define MM_ERROR          2
#define MM_WARNING        3
#define MM_INFO           4

#define MM_NULLLBL        ((char *) NULL)
#define MM_NULLSEV        MM_NOSEV
#define MM_NULLMC         MM_NULL
#define MM_NULLTXT        ((char *) NULL)
#define MM_NULLACT        ((char *) NULL)
#define MM_NULLTAG        ((char *) NULL)

#define MM_NOTOK          -1
#define MM_OK             0x00
#define MM_NOMSG          0x01
#define MM_NOCON          0x04
```

Figure 6-26: <fnmatch.h>*

```
#define FNM_PATHNAME      0x001
#define FNM_PERIOD        0x002
#define FNM_NOESCAPE      0x004
#define FNM_BADRANGE      0x008
#define FNM_EXTENDED      0x020

#define FNM_NOSYS         (-1)
#define FNM_NOMATCH       (-2)
```

Figure 6-27: <ftw.h>

```
#define FTW_F             0
#define FTW_D             1
#define FTW_DNR           2
#define FTW_NS            3
#define FTW_SL            4
#define FTW_DP            6
#define FTW_SLN           7

#define FTW_PHYS          01
#define FTW_MOUNT         02
#define FTW_CHDIR         04
#define FTW_DEPTH         010

struct FTW {
      int   quit;
      int   base;
      int   level;
};

#define FTW_SKD           1
#define FTW_FOLLOW        2
#define FTW_PRUNE         4
```

Figure 6-28: `<glob.h>*`

```
#define GLOB_APPEND        0x0001
#define GLOB_DOOFFS        0x0002
#define GLOB_ERR           0x0004
#define GLOB_MARK          0x0008
#define GLOB_NOCHECK       0x0010
#define GLOB_NOSORT        0x0020
#define GLOB_NOESCAPE      0x0040
#define GLOB_OKAYDOT       0x0200
#define GLOB_BADRANGE      0x0400
#define GLOB_EXTENDED      0x1000

#define GLOB_NOSYS         (-1)
#define GLOB_ABORTED       (-2)
#define GLOB_NOSPACE       (-3)
#define GLOB_NOMATCH       (-4)

typedef struct
{
      void         *;
      char         **gl_pathv;
      size_t       gl_pathc;
      size_t       gl_offs;
} glob_t;
```

Figure 6-29: <grp.h>

```
struct group {
       char   *gr_name;
       char   *gr_passwd;
       gid_t  gr_gid;
       char   **gr_mem;
};
```

Figure 6-30: <iconv.h>*

```
typedef void *iconv_t;
```

Figure 6-31: <sys/ipc.h>

```
struct ipc_perm {
      uid_t        uid;
      gid_t        gid;
      uid_t        cuid;
      gid_t        cgid;
      mode_t       mode;
      ulong        seq;
      key_t        key;
      long         pad[4];
};

#define IPC_CREAT   0001000
#define IPC_EXCL    0002000
#define IPC_NOWAIT  0004000

#define IPC_PRIVATE (key_t)0

#define IPC_RMID          10
#define IPC_SET           11
#define IPC_STAT          12
```

Figure 6-32: <langinfo.h>**, Part 1 of 2**

```
#define DAY_1               1
#define DAY_2               2
#define DAY_3               3
#define DAY_4               4
#define DAY_5               5
#define DAY_6               6
#define DAY_7               7

#define ABDAY_1             8
#define ABDAY_2             9
#define ABDAY_3             10
#define ABDAY_4             11
#define ABDAY_5             12
#define ABDAY_6             13
#define ABDAY_7             14

#define MON_1               15
#define MON_2               16
#define MON_3               17
#define MON_4               18
#define MON_5               19
#define MON_6               20
#define MON_7               21
#define MON_8               22
#define MON_9               23
#define MON_10              24
#define MON_11              25
#define MON_12              26
```

Figure 6-33: <langinfo.h>**, Part 2 of 2**

```
#define ABMON_1          27
#define ABMON_2          28
#define ABMON_3          29
#define ABMON_4          30
#define ABMON_5          31
#define ABMON_6          32
#define ABMON_7          33
#define ABMON_8          34
#define ABMON_9          35
#define ABMON_10         36
#define ABMON_11         37
#define ABMON_12         38

#define RADIXCHAR        39
#define THOUSEP          40
#define YESSTR           41
#define NOSTR            42
#define CRNCYSTR         43

#define D_T_FMT          44
#define D_FMT            45
#define T_FMT            46
#define AM_STR           47
#define PM_STR           48
#define CODESET          49                    M
#define T_FMT_AMPM       50                    M
#define ERA              51                    M
#define ERA_D_FMT        52                    M
#define ERA_D_T_FMT      53                    M
#define ERA_T_FMT        54                    M
#define ALT_DIGITS       55                    M
#define YESEXPR          56                    M
#define NOEXPR           57                    M
```

Figure 6-34: `<limits.h>`**, Part 1 of 2**

```
#define CHAR_BIT          8
#define SCHAR_MIN         (-128)
#define SCHAR_MAX         127
#define UCHAR_MAX         255
#define MB_LEN_MAX        5

#define CHAR_MIN          SCHAR_MIN
#define CHAR_MAX          SCHAR_MAX

#define SHRT_MIN          (-32768)
#define SHRT_MAX          32767
#define USHRT_MAX         65535
#define INT_MIN           (-2147483647-1)
#define INT_MAX           2147483647
#define UINT_MAX          4294967295
#define LONG_MIN          (-2147483647-1)
#define LONG_MAX          2147483647
#define ULONG_MAX         4294967295

#define ARG_MAX           *
#define LINK_MAX          *
#define MAX_CANON         *
#define MAX_INPUT         *
#define NGROUPS_MAX       *
#define PATH_MAX          *
#define PIPE_BUF          *
#define TMP_MAX           *
#define PASS_MAX          *
#define CHILD_MAX         *

/* starred values vary and should be
      retrieved using sysconf() or pathconf() */
```

Figure 6-35: `<limits.h>`**, Part 2 of 2**

```
#define NL_ARGMAX          9
#define NL_LANGMAX         14
#define NL_MSGMAX          32767
#define NL_NMAX            1
#define NL_SETMAX          255
#define NL_TEXTMAX         255
#define NZERO              20

#define WORD_BIT           32
#define LONG_BIT           32

#define DBL_DIG            15
#define DBL_MAX            1.7976931348623157E+308
#define DBL_MIN            2.2250738585072014E-308
#define FLT_DIG            6
#define FLT_MAX            3.40282347E+38F
#define FLT_MIN            1.17549435E-38F

#define FCHR_MAX           1048576
```

Figure 6-36: <locale.h>

```
struct lconv {
      char   *decimal_point;
      char   *thousands_sep;
      char   *grouping;
      char   *int_curr_symbol;
      char   *currency_symbol;
      char   *mon_decimal_point;
      char   *mon_thousands_sep;
      char   *mon_grouping;
      char   *positive_sign;
      char   *negative_sign;
      char   int_frac_digits;
      char   frac_digits;
      char   p_cs_precedes;
      char   p_sep_by_space;
      char   n_cs_precedes;
      char   n_sep_by_space;
      char   p_sign_posn;
      char   n_sign_posn;
};

#define LC_CTYPE          0
#define LC_NUMERIC        1
#define LC_TIME           2
#define LC_COLLATE        3
#define LC_MONETARY       4
#define LC_MESSAGES       5
#define LC_ALL            6
```

Figure 6-37: <lwpsynch.h>*

```
typedef volatile struct {
      char         wanted;
      _simplelock_t       lock;
} lwp_mutex_t;

typedef volatile struct {
      char         wanted;
} lwp_cond_t;
```

Figure 6-38: <machlock.h>*

```
typedef volatile unsigned char _simplelock_t;
```

Figure 6-39: <math.h>

```
extern const double __huge_val;
#define HUGE_VAL    __huge_val
```

G
G

Figure 6-40: <sys/mman.h>

```
#define PROT_READ         0x1
#define PROT_WRITE        0x2
#define PROT_EXEC         0x4
#define PROT_NONE         0x0

#define MAP_SHARED        1
#define MAP_PRIVATE       2
#define MAP_FIXED         0x10

#define MS_SYNC           0x0
#define MS_ASYNC          0x1
#define MS_INVALIDATE     0x2

#define PROC_TEXT   (PROT_EXEC | PROT_READ)                    G
#define PROC_DATA   (PROT_READ | PROT_WRITE | PROT_EXEC)       G

#define SHARED            0x10                                 G
#define PRIVATE           0x20                                 G

#define      MC_SYNC            1                             G
#define      MC_LOCK            2                             G
#define      MC_UNLOCK    3                                    G
#define      MC_LOCKAS    5                                    G
#define      MC_UNLOCKAS  6                                    G

#define      MCL_CURRENT  0x1                                  G
#define      MCL_FUTURE   0x2                                  G
```

Figure 6-41: <sys/mod.h>*

```
#define VOID              void
#define MAXPATHLEN        1024
#define MODMAXLINKINFOLEN 32

struct modspecific_stat {
      char   mss_linkinfo[MODMAXLINKINFOLEN];
      int    mss_type;
      int    mss_p0[2];
      int    mss_p1[2];
};

#define MODMAXLINK         4

struct modstatus {
      int                ms_id;
      VOID               *ms_base;
      unsigned int       ms_size;
      int                ms_rev;
      char               ms_path[MAXPATHLEN];
      time_t             ms_unload_delay;
      int                ms_refcnt;
      int                ms_depcnt;
      struct modspecific_stat   ms_msinfo[MODMAXLINK];
};
```

Figure 6-42: <sys/mount.h>

```
#define MS_RDONLY          0x01
#define MS_FSS             0x02
#define MS_DATA            0x04
#define MS_HADBAD          0x08

#define MS_NOSUID          0x10
#define MS_REMOUNT         0x20
#define MS_NOTRUNC         0x40
```

Figure 6-43: <sys/msg.h>

```
#define MSG_NOERROR       010000

struct msqid_ds {
      struct ipc_perm    msg_perm;
      struct msg   *msg_first;
      struct msg   *msg_last;
      ulong        msg_cbytes;
      ulong        msg_qnum;
      ulong        msg_qbytes;
      pid_t        msg_lspid;
      pid_t        msg_lrpid;
      time_t       msg_stime;
      long         msg_pad1;
      time_t       msg_rtime;
      long         msg_pad2;
      time_t       msg_ctime;
      long         msg_pad3;
      long         msg_pad4[4];
};

struct msg {
      struct msg   *msg_next;
      long         msg_type;
      ushort       msg_ts;
      short        msg_spot;
};
```

Figure 6-44: <netconfig.h>**, Part 1 of 2**

G

```
struct netconfig {
      char              *nc_netid;
      unsigned long     nc_semantics;
      unsigned long     nc_flag;
      char              *nc_protofmly;
      char              *nc_proto;
      char              *nc_device;
      unsigned long     nc_nlookups;
      char              **nc_lookups;
      unsigned long     nc_unused[8];
};

#define NC_TPI_CLTS             1
#define NC_TPI_COTS             2
#define NC_TPI_COTS_ORD         3
#define NC_TPI_RAW              4

#define NC_NOFLAG               00
#define NC_VISIBLE              01
```

Figure 6-45: <netconfig.h>**, Part 2 of 2**

```
#define NC_NOPROTOFMLY     "-"
#define NC_LOOPBACK        "loopback"
#define NC_INET            "inet"
#define NC_IMPLINK         "implink"
#define NC_PUP             "pup"
#define NC_CHAOS           "chaos"
#define NC_NS              "ns"
#define NC_NBS             "nbs"
#define NC_ECMA            "ecma"
#define NC_DATAKIT         "datakit"
#define NC_CCITT           "ccitt"
#define NC_SNA             "sna"
#define NC_DECNET          "decnet"
#define NC_DLI             "dli"
#define NC_LAT             "lat"
#define NC_HYLINK          "hylink"
#define NC_APPLETALK       "appletalk"
#define NC_NIT             "nit"
#define NC_IEEE802         "ieee802"
#define NC_OSI             "osi"
#define NC_X25             "x25"
#define NC_OSINET          "osinet"
#define NC_GOSIP           "gosip"
#define NC_NETWARE         "netware"                M

#define NC_NOPROTO         "-"
#define NC_TCP             "tcp"
#define NC_UDP             "udp"
#define NC_ICMP            "icmp"
#define NC_IPX             "ipx"                    M
#define NC_SPX             "spx"                    M
```

Figure 6-46: <netdir.h>**, Part 1 of 2**

```
struct nd_addrlist {
      int                n_cnt;
      struct netbuf      *n_addrs;
};

struct nd_hostservlist {
      int                h_cnt;
      struct nd_hostserv  *h_hostservs;
};

struct nd_hostserv {
      char         *h_host;
      char         *h_serv;
};

#define ND_HOSTSERV             0
#define ND_HOSTSERVLIST         1
#define ND_ADDR                 2
#define ND_ADDRLIST             3

#define ND_BADARG               -2
#define ND_NOMEM                -1
#define ND_OK                   0
#define ND_NOHOST               1
#define ND_NOSERV               2
#define ND_NOSYM                3
#define ND_OPEN                 4
#define ND_ACCESS               5
#define ND_UKNWN                6
#define ND_NOCTRL               7
#define ND_FAILCTRL             8
#define ND_SYSTEM               9
#define ND_NOERRMEM             10                  M
#define ND_NOLIB                11                  M
#define ND_XTIERROR             12                  M
#define ND_BADSTATE             13                  M
```

Figure 6-47: <netdir.h>**, Part 2 of 2**

```
#define ND_SET_BROADCAST          1
#define ND_SET_RESERVEDPORT       2
#define ND_CHECK_RESERVEDPORT     3
#define ND_MERGEADDR              4
#define ND_CLEAR_BROADCAST        5                    M
#define ND_SET_REUSEADDR          6                    M
#define ND_CLEAR_REUSEADDR        7                    M

#define HOST_SELF                 "\\1"
#define HOST_ANY                  "\\2"
#define HOST_BROADCAST            "\\3"
```

Figure 6-48: <nl_types.h>

```
#define NL_SETD           1

typedef int  nl_item;
typedef void *nl_catd;
```

Figure 6-49: <sys/param.h>

```
#define CANBSIZ           256
#define HZ                100
#define TICK              10000000

#define NGROUPS_UMIN      0

#define NBPSCTR           512

#define MAXPATHLEN        1024
#define MAXSYMLINKS       20
#define MAXNAMELEN        256

#define NADDR             13

#define PIPE_MAX          5120

#define NBBY              8

#define MAXFRAG           8
```

Figure 6-50: <poll.h>

```
struct pollfd {
      int    fd;
      short  events;
      short  revents;
};

#define POLLIN            0x0001
#define POLLPRI           0x0002
#define POLLOUT           0x0004
#define POLLRDNORM        0x0040
#define POLLWRNORM        POLLOUT
#define POLLRDBAND        0x0080
#define POLLWRBAND        0x0100

#define POLLNORM          POLLRDNORM

#define POLLERR           0x0008
#define POLLHUP           0x0010
#define POLLNVAL          0x0020
```

Figure 6-51: <sys/priocntl.h>*

```
#define PC_GETCID         0
#define PC_GETCLINFO            1
#define PC_SETPARMS       2
#define PC_GETPARMS       3

#define PC_CLNULL         -1

#define PC_CLNMSZ   16
#define PC_CLINFOSZ (32 / sizeof(long))
#define PC_CLPARMSZ (32 / sizeof(long))

typedef struct pcinfo {
      id_t   pc_cid;
      char   pc_clname[PC_CLNMSZ];
      long   pc_clinfo[PC_CLINFOSZ];
} pcinfo_t;

typedef struct pcparms {
      id_t   pc_cid;
      long   pc_clparms[PC_CLPARMSZ];
} pcparms_t;
```

Figure 6-52: <sys/procset.h>

```
#define P_INITPID         1
#define P_INITUID         0
#define P_INITPGID        0

typedef enum idtype {
      P_PID,
      P_PPID,
      P_PGID,
      P_SID,
      P_CID,
      P_UID,
      P_GID,
      P_ALL
} idtype_t;

typedef enum idop {
      POP_DIFF,
      POP_AND,
      POP_OR,
      POP_XOR
} idop_t;


typedef struct procset {
      idop_t       p_op;
      idtype_t     p_lidtype;
      id_t         p_lid;
      idtype_t     p_ridtype;
      id_t         p_rid;
} procset_t;
```

Figure 6-53: <pwd.h>

```
struct passwd {
      char          *pw_name;
      char          *pw_passwd;
      uid_t         pw_uid;
      gid_t         pw_gid;
      char          *pw_age;
      char          *pw_comment;
      char          *pw_gecos;
      char          *pw_dir;
      char          *pw_shell;
};
```

Figure 6-54: <regex.h>*, Part 1 of 2

```
#define REG_NOTBOL         0x000001
#define REG_NOTEOL         0x000002
#define REG_NONEMPTY       0x000004

#define REG_OR             0x000001
#define REG_PLUS           0x000002
#define REG_QUEST          0x000004
#define REG_BRACES         0x000008
#define REG_PARENS         0x000010
#define REG_ANCHORS        0x000020
#define REG_NOBACKREF      0x000040
#define REG_NOAUTOQUOTE    0x000080

#define REG_EXTENDED       (REG_OR | REG_PLUS | REG_QUEST |
             REG_BRACES | REG_PARENS | REG_ANCHORS |
             REG_NOBACKREF | REG_NOAUTOQUOTE)
#define REG_ICASE          0x000100
#define REG_NOSUB          0x000200
#define REG_NEWLINE        0x000400
#define REG_ONESUB         0x000800
#define REG_BADRANGE       0x004000
#define REG_ANGLES         0x040000
#define REG_ESCNL          0x080000
#define REG_OLDBRE  (REG_BADRANGE | REG_ANGLES | REG_ESCNL)
```


DRAFT COPY
March 19, 1997
File: abi_386/chap6 (Delta 44.21)
386:adm.book:sum

Figure 6-55: <regex.h>*, Part 2 of 2

```
#define REG_ENOSYS        (-1)
#define REG_NOMATCH       1
#define REG_BADPAT        2
#define REG_ECOLLATE      3
#define REG_ECTYPE        4
#define REG_EESCAPE       7
#define REG_ESUBREG       8
#define REG_EBRACK        9
#define REG_NOPAT         12
#define REG_EPAREN        13
#define REG_EBRACE        14
#define REG_BADBR         15
#define REG_ERANGE        16
#define REG_ESPACE        17
#define REG_BADRPT        18

typedef struct
{
      size_t       re_nsub;
      unsigned long      re_flags;
      void         *[4];
} regex_t;

typedef ssize_t regoff_t;

typedef struct
{
      regoff_t     rm_so;
      regoff_t     rm_eo;
} regmatch_t;
```

Figure 6-56: <sys/resource.h>

```
#define RLIMIT_CPU        0
#define RLIMIT_FSIZE      1
#define RLIMIT_DATA       2
#define RLIMIT_STACK      3
#define RLIMIT_CORE       4
#define RLIMIT_NOFILE     5
#define RLIMIT_VMEM       6
#define RLIM_NLIMITS      7
#define RLIMIT_AS         RLIMIT_VMEM
#define RLIM_INFINITY     0x7fffffff

typedef unsigned long     rlim_t;

struct rlimit {
      rlim_t       rlim_cur;
      rlim_t       rlim_max;
};
```

Figure 6-57: <rpc.h>**, Part 1 of 16**

```
#define bool_t            int
#define enum_t            int

enum xdr_op {
       XDR_ENCODE=0,
       XDR_DECODE=1,
       XDR_FREE=2
};

typedef bool_t     (*xdrproc_t)();

typedef struct {
       enum xdr_op  x_op;
       struct xdr_ops {
              bool_t (*x_getlong)();
              bool_t (*x_putlong)();
              bool_t (*x_getbytes)();
              bool_t (*x_putbytes)();
              u_int  (*x_getpostn)();
              bool_t (*x_setpostn)();
              long * (*x_inline)();
              void   (*x_destroy)();
       } *x_ops;
       caddr_t      x_public;
       caddr_t      x_private;
       caddr_t      x_base;
       int          x_handy;
} XDR;
```

Figure 6-58: <rpc.h>**, Part 2 of 16**

```
#define xdr_getpos(xdrs)                                \
      (*(xdrs)->x_ops->x_getpostn)(xdrs)
#define xdr_setpos(xdrs, pos)                           \
      (*(xdrs)->x_ops->x_setpostn)(xdrs, pos)
#define xdr_inline(xdrs, len)                           \
      (*(xdrs)->x_ops->x_inline)(xdrs, len)
#define xdr_destroy(xdrs)                          \          M
      (*(xdrs)->x_ops->x_destroy)(xdrs)                       M

#define NULL_xdrproc_t    ((xdrproc_t)0)
struct xdr_discrim {
      int          value;
      xdrproc_t    proc;
};
```

Figure 6-59: `<rpc.h>`**, Part 3 of 16**

```
#define MAX_AUTH_BYTES          400
#define MAXNETNAMELEN           255
#define HEXKEYBYTES             48

enum auth_stat {
       AUTH_OK=0,
       AUTH_BADCRED=1,
       AUTH_REJECTEDCRED=2,
       AUTH_BADVERF=3,
       AUTH_REJECTEDVERF=4,
       AUTH_TOOWEAK=5,
       AUTH_INVALIDRESP=6,
       AUTH_FAILED=7
};

typedef u_long      u_int32                                  M

union des_block {
       struct {
              u_int32            high;                       M
              u_int32            low;                        M
       } key;
       char   c[8];
};
typedef union des_block         des_block;
```


DRAFT COPY
March 19, 1997
File: abi_386/chap6 (Delta 44.21)
386:adm.book:sum

Figure 6-60: <rpc.h>**, Part 4 of 16**

```
struct opaque_auth {
       enum_t       oa_flavor;
       caddr_t      oa_base;
       u_int        oa_length;
};

typedef struct {
       struct opaque_auth  ah_cred;
       struct opaque_auth  ah_verf;
       union des_block     ah_key;
       struct auth_ops {
              void  (*ah_nextverf)();
              int   (*ah_marshal)();
              int   (*ah_validate)();
              int   (*ah_refresh)();
              void  (*ah_destroy)();
       } *ah_ops;
       caddr_t      ah_private;
} AUTH;

#define auth_destroy(auth)      \
              ((*((auth)->ah_ops->ah_destroy))(auth))

#define AUTH_NONE         0
#define AUTH_NULL         0
#define AUTH_SYS          1
#define AUTH_UNIX         AUTH_SYS
#define AUTH_SHORT        2
#define AUTH_DES          3
#define AUTH_ESV          200004                    M
```

Figure 6-61: <rpc.h>**, Part 5 of 16**

```
enum clnt_stat {
       RPC_SUCCESS=0,
       RPC_CANTENCODEARGS=1,
       RPC_CANTDECODERES=2,
       RPC_CANTSEND=3,
       RPC_CANTRECV=4,
       RPC_TIMEDOUT=5,
       RPC_INTR=18,
       RPC_UDERROR=23,
       RPC_VERSMISMATCH=6,
       RPC_AUTHERROR=7,
       RPC_PROGUNAVAIL=8,
       RPC_PROGVERSMISMATCH=9,
       RPC_PROCUNAVAIL=10,
       RPC_CANTDECODEARGS=11,
       RPC_SYSTEMERROR=12,
       RPC_UNKNOWNHOST=13,
       RPC_UNKNOWNPROTO=17,
       RPC_UNKNOWNADDR=19,
       RPC_NOBROADCAST=21,
       RPC_RPCBFAILURE=14,
       RPC_PROGNOTREGISTERED=15,
       RPC_N2AXLATEFAILURE=22,
       RPC_TLIERROR=20,
       RPC_FAILED=16
};
#define RPC_PMAPFAILURE    RPC_RPCBFAILURE
```

Figure 6-62: <rpc.h>, **Part 6 of 16**

```
struct rpc_err {
       enum clnt_stat      re_status;
       union {
              struct {
                     int    errno;
                     int    t_errno;
              } RE_err;
              enum auth_stat      RE_why;
              struct {
                     u_long       low;
                     u_long       high;
              } RE_vers;
              struct {
                     long   s1;
                     long   s2;
              } RE_lb;
       } ru;
};
typedef struct {
       AUTH   *cl_auth;
       struct clnt_ops {
              enum clnt_stat      (*cl_call)();
              void         (*cl_abort)();
              void         (*cl_geterr)();
              bool_t       (*cl_freeres)();
              void         (*cl_destroy)();
              bool_t       (*cl_control)();
       } *cl_ops;
       caddr_t             cl_private;
       char                *cl_netid;
       char                *cl_tp;
} CLIENT;
```

Figure 6-63: <rpc.h>**, Part 7 of 16**

```
#define FEEDBACK_REXMIT1  1
#define FEEDBACK_OK       2

#define clnt_call(rh, proc, xargs, argsp, xres, resp, secs)\
      ((*(rh)->cl_ops->cl_call)                    \
      (rh, proc, xargs, argsp, xres, resp, secs))
#define clnt_abort(rh)                             \
      ((*(rh)->cl_ops->cl_abort)(rh))
#define clnt_geterr(rh, errp)                      \
      ((*(rh)->cl_ops->cl_geterr)(rh, errp))
#define clnt_freeres(rh,xres,resp)                 \
      ((*(rh)->cl_ops->cl_freeres)(rh,xres,resp))
#define clnt_control(cl, rq, in)                   \
      ((*(cl)->cl_ops->cl_control)(cl, rq, in))
#define clnt_destroy(rh)                           \
      ((*(rh)->cl_ops->cl_destroy)(rh))

#define CLSET_TIMEOUT           1
#define CLGET_TIMEOUT           2
#define CLGET_SERVER_ADDR       3
#define CLGET_FD                6
#define CLGET_SVC_ADDR          7
#define CLSET_FD_CLOSE          8
#define CLSET_FD_NCLOSE         9
#define CLSET_RETRY_TIMEOUT     4
#define CLGET_RETRY_TIMEOUT     5
```

Figure 6-64: <rpc.h>**, Part 8 of 16**

```
typedef struct {                                        M
        enum    clnt_stat       cf_stat;                M
        struct  rpc_err         cf_error;               M
} rpc_createerr_t;                                      M

extern  rpc_createerr_t         rpc_createerr;          M
```

Figure 6-65: <rpc.h>**, Part 9 of 16**

```
enum msg_type {
       CALL=0,
       REPLY=1
};

enum reply_stat {
       MSG_ACCEPTED=0,
       MSG_DENIED=1
};

enum accept_stat {
       SUCCESS=0,
       PROG_UNAVAIL=1,
       PROG_MISMATCH=2,
       PROC_UNAVAIL=3,
       GARBAGE_ARGS=4,
       SYSTEM_ERR=5
};

enum reject_stat {
       RPC_MISMATCH=0,
       AUTH_ERROR=1
};
```

Figure 6-66: <rpc.h>, **Part 10 of 16**

```
struct accepted_reply {
       struct opaque_auth        ar_verf;
       enum accept_stat          ar_stat;
       union {
              struct {
                     u_long       low;
                     u_long       high;
              } AR_versions;
              struct {
                     caddr_t      where;
                     xdrproc_t    proc;
              } AR_results;

       } ru;
};

struct rejected_reply {
       enum reject_stat    rj_stat;
       union {
              struct {
                     u_long       low;
                     u_long       high;
              } RJ_versions;
              enum auth_stat      RJ_why;
       } ru;
};
```

Figure 6-67: <rpc.h>**, Part 11 of 16**

```
struct reply_body {
       enum reply_stat     rp_stat;
       union {
              struct accepted_reply RP_ar;
              struct rejected_reply RP_dr;
       } ru;
};

struct call_body {
       u_long              cb_rpcvers;
       u_long              cb_prog;
       u_long              cb_vers;
       u_long              cb_proc;
       struct opaque_auth  cb_cred;
       struct opaque_auth  cb_verf;
};

struct rpc_msg {
       u_long              rm_xid;
       enum msg_type       rm_direction;
       union {
              struct call_body    RM_cmb;
              struct reply_body   RM_rmb;
       } ru;
};
```

Figure 6-68: <rpc.h>, **Part 12 of 16**

```
struct authsys_parms {
       u_long       aup_time;
       char         *aup_machname;
       uid_t        aup_uid;
       gid_t        aup_gid;
       u_int        aup_len;
       gid_t        *aup_gids;
};
```

Figure 6-69: <rpc.h>, **Part 13 of 16**

```
enum authdes_namekind {
       ADN_FULLNAME,
       ADN_NICKNAME
};

struct authdes_fullname {
       char                *name;
       des_block           key;
       u_long              window;
};

struct authdes_cred {
       enum authdes_namekind     adc_namekind;
       struct authdes_fullname   adc_fullname;
       u_long                    adc_nickname;
};
```

Figure 6-70: <rpc.h>**, Part 14 of 16**

```
enum xprt_stat {
       XPRT_DIED,
       XPRT_MOREREQS,
       XPRT_IDLE
};

typedef struct {
       int                 xp_fd;
       u_short             xp_port;
       struct xp_ops {
              bool_t       (*xp_recv)();
              enum xprt_stat     (*xp_stat)();
              bool_t       (*xp_getargs)();
              bool_t       (*xp_reply)();
              bool_t       (*xp_freeargs)();
              void         (*xp_destroy)();
       } *xp_ops;
       int                 xp_addrlen;
       char                *xp_tp;
       char                *xp_netid;
       struct netbuf       xp_ltaddr;
       struct netbuf       xp_rtaddr;
       char                xp_raddr[16];
       struct opaque_auth  xp_verf;
       caddr_t             xp_p1;
       caddr_t             xp_p2;
       caddr_t             xp_p3;
       int                 xp_type;
} SVCXPRT;
```

M

Figure 6-71: <rpc.h>**, Part 15 of 16**

```
#define svc_getrpccaller(x) (&(x)->xp_rtaddr)
#define svc_getargs(xprt, xargs, argsp)             \
      (*(xprt)->xp_ops->xp_getargs)((xprt), (xargs), (argsp))
#define svc_freeargs(xprt, xargs, argsp)            \
      (*(xprt)->xp_ops->xp_freeargs)((xprt), (xargs), (argsp))
#define svc_destroy(xprt)                           \
      (*(xprt)->xp_ops->xp_destroy)(xprt)

struct svc_req {
      u_long              rq_prog;
      u_long              rq_vers;
      u_long              rq_proc;
      struct opaque_auth  rq_cred;
      caddr_t             rq_clntcred;
      SVCXPRT             *rq_xprt;
};


#define FD_SETSIZE      1024                                    M
#define NBBY 8                                                  M

typedef long    fd_mask;                                        M
#define NFDBITS (sizeof(fd_mask) * NBBY)                        M
#define howmany(x, y)   (((x)+((y)-1))/(y))                     M

typedef struct fd_set {                                         M
        fd_mask fds_bits[howmany(FD_SETSIZE, NFDBITS)];         M
} fd_set;                                                       M

extern fd_set svc_fdset;
```

Figure 6-72: <rpc.h>**, Part 16 of 16**

```
struct rpcb {
      u_long       r_prog;
      u_long       r_vers;
      char         *r_netid;
      char         *r_addr;
      char         *r_owner;
};
typedef struct rpcb RPCB;

struct rpcblist {
      RPCB               rpcb_map;
      struct rpcblist    *rpcb_next;
};
```

Figure 6-73: <rtpriocntl.h>*

```
typedef struct rtparms {
      short  rt_pri;
      ulong  rt_tqsecs;
      long   rt_tqnsecs;
} rtparms_t;

typedef struct rtinfo {
      short  rt_maxpri;
} rtinfo_t;

#define RT_NOCHANGE -1
#define RT_TQINF    -2
#define RT_TQDEF    -3
```

Figure 6-74: <search.h>

```
typedef enum { FIND, ENTER } ACTION;
typedef struct entry { char *key; void *data; } ENTRY;
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

Figure 6-75: <sys/sem.h>

```
#define SEM_UNDO    010000
#define GETNCNT     3
#define GETPID      4
#define GETVAL      5
#define GETALL      6
#define GETZCNT     7
#define SETVAL      8
#define SETALL      9

struct semid_ds {
       struct ipc_perm     sem_perm;
       struct sem   *sem_base;
       ushort       sem_nsems;
       time_t       sem_otime;
       long         sem_pad1;
       time_t       sem_ctime;
       long         sem_pad2;
       long         sem_pad3[4];
};

struct sem {
       ushort       semval;
       pid_t        sempid;
       ushort       semncnt;
       ushort       semzcnt;
};

struct sembuf {
       ushort       sem_num;
       short        sem_op;
       short        sem_flg;
};
```

Figure 6-76: <setjmp.h>

```
#define _SIGJBLEN         128
#define _JBLEN            10

typedef int  jmp_buf[_JBLEN];
typedef int  sigjmp_buf[_SIGJBLEN];
```

Figure 6-77: <sys/shm.h>

```
#define SHMLBA            ((1)<<12)

#define SHM_RDONLY        010000
#define SHM_RND           020000

struct shmid_ds {
      struct ipc_perm     shm_perm;
      int           shm_segsz;
      struct anon_map     *shm_amp;
      ushort        shm_lkcnt;
      pid_t         shm_lpid;
      pid_t         shm_cpid;
      ulong         shm_nattch;
      ulong         shm_cnattch;
      time_t        shm_atime;
      long          shm_pad1;
      time_t        shm_dtime;
      long          shm_pad2;
      time_t        shm_ctime;
      long          shm_pad3;
      long          shm_pad4[4];
};
```

Figure 6-78: `<signal.h>`**, Part 1 of 3**

```
#define SIGHUP                  1
#define SIGINT                  2
#define SIGQUIT                 3
#define SIGILL                  4
#define SIGTRAP                 5
#define SIGIOT                  6
#define SIGABRT                 6
#define SIGEMT                  7
#define SIGFPE                  8
#define SIGKILL                 9
#define SIGBUS                  10
#define SIGSEGV                 11
#define SIGSYS                  12
#define SIGPIPE                 13
#define SIGALRM                 14
#define SIGTERM                 15
#define SIGUSR1                 16
#define SIGUSR2                 17
#define SIGCLD                  18
#define SIGCHLD                 18
#define SIGPWR                  19
#define SIGWINCH                20
#define SIGURG                  21
#define SIGPOLL                 22
#define SIGIO                   22
#define SIGSTOP                 23
#define SIGTSTP                 24
#define SIGCONT                 25
#define SIGTTIN                 26
#define SIGTTOU                 27
#define SIGVTALRM               28
#define SIGPROF                 29
```

Figure 6-79: <signal.h>**, Part 2 of 3**

```
#define SIGXCPU                 30
#define SIGXFSZ                 31

#define SIG_DFL           (void(*)())0
#define SIG_ERR           (void(*)())-1
#define SIG_IGN           (void(*)())1
#define SIG_HOLD          (void(*)())2

#define SIG_BLOCK         1
#define SIG_UNBLOCK       2
#define SIG_SETMASK       3

typedef struct {                                    M
      unsigned int        sa_sigbits[4];            M
} sigset_t;                                         M

struct sigaction {
      int           sa_flags;
      void          (*sa_handler)();
      sigset_t      sa_mask;
      int           sa_resv[2];
};

#define SA_NOCLDSTOP      0x00020000
#define SA_ONSTACK        0x00000001
#define SA_RESETHAND      0x00000002
#define SA_RESTART        0x00000004
#define SA_SIGINFO        0x00000008
#define SA_NODEFER        0x00000010
#define SA_NOCLDWAIT      0x00010000
```

Figure 6-80: <signal.h>**, Part 3 of 3**

```
#define SS_ONSTACK        0x00000001
#define SS_DISABLE        0x00000002

struct sigaltstack {
      char   *ss_sp;
      int    ss_size;
      int    ss_flags;
};

typedef struct sigaltstack      stack_t;
```

Figure 6-81: <sys/siginfo.h>**, Part 1 of 5**

```
#define ILL_ILLOPC        1
#define ILL_ILLOPN        2
#define ILL_ILLADR        3
#define ILL_ILLTRP        4
#define ILL_PRVOPC        5
#define ILL_PRVREG        6
#define ILL_COPROC        7
#define ILL_BADSTK        8

#define FPE_INTDIV        1
#define FPE_INTOVF        2
#define FPE_FLTDIV        3
#define FPE_FLTOVF        4
#define FPE_FLTUND        5
#define FPE_FLTRES        6
#define FPE_FLTINV        7
#define FPE_FLTSUB        8
```

Figure 6-82: <sys/siginfo.h>**, Part 2 of 5**

```
#define SEGV_MAPERR       1
#define SEGV_ACCERR       2

#define BUS_ADRALN        1
#define BUS_ADRERR        2
#define BUS_OBJERR        3

#define TRAP_BRKPT        1
#define TRAP_TRACE        2

#define CLD_EXITED        1
#define CLD_KILLED        2
#define CLD_DUMPED        3
#define CLD_TRAPPED       4
#define CLD_STOPPED       5
#define CLD_CONTINUED     6

#define POLL_IN           1
#define POLL_OUT          2
#define POLL_MSG          3
#define POLL_ERR          4
#define POLL_PRI          5
#define POLL_HUP          6

#define SI_MAXSZ          128
#define SI_PAD            ((SI_MAXSZ / sizeof(int)) - 3)
```

Figure 6-83: <sys/siginfo.h>**, Part 3 of 5**

```
typedef struct siginfo {
       int    si_signo;
       int    si_code;
       int    si_errno;
       union {
              int    _pad[SI_PAD];
              struct {
                     pid_t  _pid;
                     union {
                            struct {
                                   uid_t  _uid;
                            } _kill;
                            struct {
                                   clock_t      _utime;
                                   int          _status;
                                   clock_t      _stime;
                            } _cld;
                     } _pdata;
              } _proc;
              struct {
                     caddr_t      _addr;
              } _fault;
              struct {
                     int    _fd;
                     long   _band;
              } _file;
       } _data;
} siginfo_t;
```

Figure 6-84: <sys/siginfo.h>**, Part 4 of 5**

```
#define si_pid            _data._proc._pid
#define si_status         _data._proc._pdata._cld._status
#define si_stime          _data._proc._pdata._cld._stime
#define si_utime          _data._proc._pdata._cld._utime
#define si_uid            _data._proc._pdata._kill._uid
#define si_addr           _data._fault._addr
#define si_fd             _data._file._fd
#define si_band           _data._file._band
```

Figure 6-85: <sys/siginfo.h>*, Part 5 of 5

```
union sigval {
        int     sival_int;
        void    *sival_ptr;
};

union notifyinfo {
        int             nisigno;
        void            (*nifunc)(union sigval);
};

struct sigevent {
        int             sigev_notify;
        union notifyinfo sigev_notifyinfo;
        union sigval    sigev_value;
};

#define SIGEV_NONE      1
#define SIGEV_SIGNAL    2
#define SIGEV_CALLBACK  3
```

Figure 6-86: <sys/stat.h>**, Part 1 of 2**

```
#define _ST_FSTYPSZ       16

struct stat {
       dev_t        st_dev;
       long         st_pad1[3];
       ino_t        st_ino;
       mode_t       st_mode;
       nlink_t      st_nlink;
       uid_t        st_uid;
       gid_t        st_gid;
       dev_t        st_rdev;
       long         st_pad2[2];
       off_t        st_size;
       long         st_pad3;
       timestruc_t  st_atim;
       timestruc_t  st_mtim;
       timestruc_t  st_ctim;
       long         st_blksize;
       long         st_blocks;
       char         st_fstype[_ST_FSTYPSZ];
       long         st_pad4[8];
};
#define st_atime    st_atim.tv_sec
#define st_mtime    st_mtim.tv_sec
#define st_ctime    st_ctim.tv_sec
```

Figure 6-87: <sys/stat.h>**, Part 2 of 2**

```
#define S_IFMT              0xF000
#define S_IFIFO             0x1000
#define S_IFCHR             0x2000
#define S_IFDIR             0x4000
#define S_IFBLK             0x6000
#define S_IFREG             0x8000
#define S_IFLNK             0xA000

#define S_ISUID             0x800
#define S_ISGID             0x400
#define S_ISVTX             0x200

#define S_IRWXU             00700
#define S_IRUSR             00400
#define S_IWUSR             00200
#define S_IXUSR             00100
#define S_IRWXG             00070
#define S_IRGRP             00040
#define S_IWGRP             00020
#define S_IXGRP             00010
#define S_IRWXO             00007
#define S_IROTH             00004
#define S_IWOTH             00002
#define S_IXOTH             00001

#define S_ISFIFO(mode)      ((mode&0xF000) == 0x1000)
#define S_ISCHR(mode)       ((mode&0xF000) == 0x2000)
#define S_ISDIR(mode)       ((mode&0xF000) == 0x4000)
#define S_ISBLK(mode)       ((mode&0xF000) == 0x6000)
#define S_ISREG(mode)       ((mode&0xF000) == 0x8000)
```

Figure 6-88: <sys/statvfs.h>

```
#define FSTYPSZ     16

typedef struct statvfs {
      u_long       f_bsize;
      u_long       f_frsize;
      u_long       f_blocks;
      u_long       f_bfree;
      u_long       f_bavail;
      u_long       f_files;
      u_long       f_ffree;
      u_long       f_favail;
      u_long       f_fsid;
      char         f_basetype[FSTYPSZ];
      u_long       f_flag;
      u_long       f_namemax;
      char         f_fstr[32];
      u_long       f_filler[16];
} statvfs_t;

#define ST_RDONLY   0x01
#define ST_NOSUID   0x02
#define ST_NOTRUNC  0x04
```

Figure 6-89: <stdarg.h>

```
typedef void                *va_list;                                   G
extern  void                 va_end(va_list);                           G
#define va_start(list, name) (void) (list = (void *)((char *)&...))     G
```