Adobe PostScript

```
struct utimbuf {
        time_t actime;          /* access time */
        time_t modtime;         /* modification time */
};
```

Figure 6-52 `<utime.h>`

```
#define    _SYS_NMLN    257

struct utsname {
        char    sysname[_SYS_NMLN];
        char    nodename[_SYS_NMLN];
        char    release[_SYS_NMLN];
        char    version[_SYS_NMLN];
        char    machine[_SYS_NMLN];
};
```

Figure 6-53 `<sys/utsname.h>`

```
#define WUNTRACED       0004
#define WNOHANG         0100
#define WEXITED         0001
#define WTRAPPED        0002
#define WSTOPPED        WUNTRACED
#define WCONTINUED      0010
#define WNOWAIT         0200
#define WOPTMASK \
          (WEXITED|WTRAPPED|WSTOPPED|WCONTINUED|WNOHANG|WNOWAIT)

#define WSTOPFLG        0177
#define WCONTFLG        0177777
#define WCOREFLG        0200
#define WSIGMASK        0177
```

Figure 6-54 `<wait.h>`

```
#define _SC_MQ_OPEN_MAX         29
#define _SC_MQ_PRIO_MAX         30
#define _SC_PRIORITIZED_IO      31
#define _SC_PRIORITY_SCHEDULING 32
#define _SC_REALTIME_SIGNALS    33
#define _SC_RTSIG_MAX           34
#define _SC_SEMAPHORES          35
#define _SC_SEM_NSEMS_MAX       36
#define _SC_SEM_VALUE_MAX       37
#define _SC_SHARED_MEMORY_OBJECTS 38
#define _SC_SIGQUEUE_MAX        39
#define _SC_SIGRT_MIN           40
#define _SC_SIGRT_MAX           41
#define _SC_SYNCHRONIZED_IO     42
#define _SC_TIMERS              43
#define _SC_TIMER_MAX           44
#define _SC_2_C_BIND            45
#define _SC_2_C_DEV             46
#define _SC_2_C_VERSION         47
#define _SC_2_FORT_DEV          48
#define _SC_2_FORT_RUN          49
#define _SC_2_LOCALEDEF         50
#define _SC_2_SW_DEV            51
#define _SC_2_UPE               52
#define _SC_2_VERSION           53
#define _SC_BC_BASE_MAX         54
#define _SC_BC_DIM_MAX          55
#define _SC_BC_SCALE_MAX        56
#define _SC_BC_STRING_MAX       57
#define _SC_COLL_WEIGHTS_MAX    58
#define _SC_EXPR_NEST_MAX       59
#define _SC_LINE_MAX            60
#define _SC_RE_DUP_MAX          61
#define _SC_XOPEN_CRYPT         62
#define _SC_XOPEN_ENH_I18N      63
#define _SC_XOPEN_SHM           64
#define _SC_PHYS_PAGES          500
#define _SC_AVPHYS_PAGES        501
#define _CS_PATH                65
#define _PC_LINK_MAX            1
#define _PC_MAX_CANON           2
#define _PC_MAX_INPUT           3
#define _PC_NAME_MAX            4
#define _PC_PATH_MAX            5
#define _PC_PIPE_BUF            6
#define _PC_NO_TRUNC            7
#define _PC_VDISABLE            8
#define _PC_CHOWN_RESTRICTED    9
#define _PC_ASYNC_IO            10
#define _PC_PRIO_IO             11
#define _PC_SYNC_IO             12
#define _PC_LAST                12
```

Figure 6-51 `<unistd.h>`

```
#define R_OK    4       /* Test for Read permission */
#define W_OK    2       /* Test for Write permission */
#define X_OK    1       /* Test for eXecute permission */
#define F_OK    0       /* Test for existence of File */

#define F_ULOCK 0       /* Unlock a previously locked region */
#define F_LOCK  1       /* Lock a region for exclusive use */
#define F_TLOCK 2      /* Test and lock a region for excl use */
#define F_TEST  3      /* Test a region for other procs locks */
#define SEEK_SET  0     /* Set file pointer to "offset" */
#define SEEK_CUR  1     /* ... to current plus "offset" */
#define SEEK_END  2     /* ... to EOF plus "offset" */

#define _POSIX_FSYNC             1
#define _POSIX_JOB_CONTROL       1
#define _POSIX_MAPPED_FILES      1
#define _POSIX_MEMLOCK           1
#define _POSIX_MEMLOCK_RANGE     1
#define _POSIX_MEMORY_PROTECTION 1
#define _POSIX_REALTIME_SIGNALS  1
#define _POSIX_SAVED_IDS         1
#define _POSIX_SYNCHRONIZED_IO   1
#define _POSIX_TIMERS            1
#define _POSIX_VDISABLE          0

#define STDIN_FILENO             0
#define STDOUT_FILENO            1
#define STDERR_FILENO            2

#define _SC_ARG_MAX              1
#define _SC_CHILD_MAX            2
#define _SC_CLK_TCK              3
#define _SC_NGROUPS_MAX          4
#define _SC_OPEN_MAX             5
#define _SC_JOB_CONTROL          6
#define _SC_SAVED_IDS            7
#define _SC_VERSION              8
#define _SC_PASS_MAX             9
#define _SC_LOGNAME_MAX         10
#define _SC_PAGESIZE            11
#define _SC_XOPEN_VERSION       12
#define _SC_NPROCESSORS_CONF    14
#define _SC_NPROCESSORS_ONLN    15
#define _SC_STREAM_MAX          16
#define _SC_TZNAME_MAX          17
#define _SC_AIO_LISTIO_MAX      18
#define _SC_AIO_MAX             19
#define _SC_AIO_PRIO_DELTA_MAX  20
#define _SC_ASYNCHRONOUS_IO     21
#define _SC_DELAYTIMER_MAX      22
#define _SC_FSYNC               23
#define _SC_MAPPED_FILES        24
#define _SC_MEMLOCK             25
#define _SC_MEMLOCK_RANGE       26
#define _SC_MEMORY_PROTECTION   27
#define _SC_MESSAGE_PASSING     28
```

```
        greg_t  r_r28;
        greg_t  r_r29;
        greg_t  r_r30;
        greg_t  r_r31;
        greg_t  r_cr;           /* Condition Register */
        greg_t  r_lr;           /* Link Register */
        greg_t  r_pc;           /* User PC (Copy of SRR0) */
        greg_t  r_msr;          /* saved MSR (Copy of SRR1) */
        greg_t  r_ctr;          /* Count Register */
        greg_t  r_xer;          /* Integer Exception Register */
        greg_t  r_mq;           /* MQ Register (601 only) */
};

typedef struct fpu {
        double      fpu_regs[32];   /* FPU regs - 32 doubles */
        unsigned    fpu_fpscr;      /* FPU status/control reg */
        unsigned    fpu_valid;      /* nonzero IFF the rest  */
                    /* of this structure contains valid data */
} fpregset_t;

typedef struct {
        gregset_t   gregs;     /* general register set */
        fpregset_t  fpregs;    /* floating point register set */
        long        filler[8];
} mcontext_t;

#define FP_NO   0   /* no fp chip or emulator (no fp support) */
#define FP_HW   1   /* fp processor present bit */

extern int fp_version;   /* kind of fp support */
extern int fpu_exists;   /* FPU hw exists      */
```

Figure 6-48 <ucontext.h>

```
typedef struct iovec {
        caddr_t iov_base;
        int     iov_len;
} iovec_t;
```

Figure 6-49 <sys/uio.h>

```
#define UL_GETFSIZE    1    /* get file limit */
#define UL_SETFSIZE    2    /* set file limit */
```

Figure 6-50 <ulimit.h>

```
typedef struct ucontext {
        unsigned long   uc_flags;
        struct ucontext *uc_link;
        sigset_t        uc_sigmask;
        stack_t         uc_stack;
        mcontext_t      uc_mcontext;
        long            uc_filler[6];
} ucontext_t;

#define GETCONTEXT      0
#define SETCONTEXT      1

#define UC_SIGMASK      001
#define UC_STACK        002
#define UC_CPU          004
#define UC_MAU          010
#define UC_FPU          UC_MAU
#define UC_INTR         020

#define UC_MCONTEXT     (UC_CPU|UC_FPU)

#define UC_ALL          (UC_SIGMASK|UC_STACK|UC_MCONTEXT)

#define NGREG   39

typedef int     greg_t;
typedef greg_t  gregset_t[NGREG];

struct regs {
        greg_t  r_r0;           /* GPRs 0 - 31 */
        greg_t  r_r1;
        greg_t  r_r2;
        greg_t  r_r3;
        greg_t  r_r4;
        greg_t  r_r5;
        greg_t  r_r6;
        greg_t  r_r7;
        greg_t  r_r8;
        greg_t  r_r9;
        greg_t  r_r10;
        greg_t  r_r11;
        greg_t  r_r12;
        greg_t  r_r13;
        greg_t  r_r14;
        greg_t  r_r15;
        greg_t  r_r16;
        greg_t  r_r17;
        greg_t  r_r18;
        greg_t  r_r19;
        greg_t  r_r20;
        greg_t  r_r21;
        greg_t  r_r22;
        greg_t  r_r23;
        greg_t  r_r24;
        greg_t  r_r25;
        greg_t  r_r26;
        greg_t  r_r27;
```

```
typedef long             time_t;
typedef long             daddr_t;
typedef unsigned long    dev_t;
typedef long             gid_t;
typedef unsigned long    ino_t;
typedef int              key_t;
typedef long             pid_t;
typedef unsigned long    mode_t;
typedef unsigned long    nlink_t;
typedef long             off_t;
typedef long             uid_t;
```

Figure 6-47 `<sys/types.h>`

```
#define T_FAKE          8      /* fake state used when state */
                              /* cannot be determined        */
#define T_HACK          12     /* T_HACK is useless but      */
                              /* exposed interface           */

#define T_NOSTATES      9

#define T_OPEN          0
#define T_BIND          1
#define T_OPTMGMT       2
#define T_UNBIND        3
#define T_CLOSE         4
#define T_SNDUDATA      5
#define T_RCVUDATA      6
#define T_RCVUDERR      7
#define T_CONNECT1      8
#define T_CONNECT2      9
#define T_RCVCONNECT    10
#define T_LISTN         11
#define T_ACCEPT1       12
#define T_ACCEPT2       13
#define T_ACCEPT3       14
#define T_SND           15
#define T_RCV           16
#define T_SNDDIS1       17
#define T_SNDDIS2       18
#define T_RCVDIS1       19
#define T_RCVDIS2       20
#define T_RCVDIS3       21
#define T_SNDREL        22
#define T_RCVREL        23
#define T_PASSCON       24

#define T_NOEVENTS      25

#define nvs             127     /* not a valid state change */

extern char tiusr_statetbl[T_NOEVENTS][T_NOSTATES];
#define LOCALNAME       0
#define REMOTENAME      1

#define TI_NORMAL       0
#define TI_EXPEDITED    1

extern char *t_errlist[];
extern int t_nerr;
```

Figure 6-46 `<sys/tiuser.h>`

```
struct t_bind {
        struct netbuf   addr;
        unsigned        qlen;
};

struct t_optmgmt {
        struct netbuf   opt;
        long            flags;
};

struct t_discon {
        struct netbuf udata;        /* user data            */
        int reason;                 /* reason code          */
        int sequence;               /* sequence number      */
};

struct t_call {
        struct netbuf addr;         /* address              */
        struct netbuf opt;          /* options              */
        struct netbuf udata;        /* user data            */
        int sequence;               /* sequence number      */
};

struct t_unitdata {
        struct netbuf addr;         /* address              */
        struct netbuf opt;          /* options              */
        struct netbuf udata;        /* user data            */
};

struct t_uderr {
        struct netbuf addr;         /* address              */
        struct netbuf opt;          /* options              */
        long    error;              /* error code           */
};

#define T_BIND          1           /* struct t_bind        */
#define T_OPTMGMT       2           /* struct t_optmgmt     */
#define T_CALL          3           /* struct t_call        */
#define T_DIS           4           /* struct t_discon      */
#define T_UNITDATA      5           /* struct t_unitdata    */
#define T_UDERROR       6           /* struct t_uderr       */
#define T_INFO          7           /* struct t_info        */

#define T_ADDR  0x01                /* address              */
#define T_OPT   0x02                /* options              */
#define T_UDATA 0x04                /* user data            */
#define T_ALL   0x07                /* all the above        */


#define T_UNINIT       0     /* uninitialized                */
#define T_UNBND        1     /* unbound                      */
#define T_IDLE         2     /* idle                         */
#define T_OUTCON       3     /* outgoing connection pending  */
#define T_INCON        4     /* incoming connection pending  */
#define T_DATAXFER     5     /* data transfer                */
#define T_OUTREL       6     /* outgoing release pending     */
#define T_INREL        7     /* incoming release pending     */
```

```
#define TBADADDR        1    /* incorrect addr format        */
#define TBADOPT         2    /* incorrect option format      */
#define TACCES          3    /* incorrect permissions        */
#define TBADF           4    /* illegal transport fd         */
#define TNOADDR         5    /* couldn't allocate addr       */
#define TOUTSTATE       6    /* out of state                 */
#define TBADSEQ         7    /* bad call sequence number     */
#define TSYSERR         8    /* system error                 */
#define TLOOK           9    /* event requires attention     */
#define TBADDATA       10    /* illegal amount of data       */
#define TBUFOVFLW      11    /* buffer not large enough      */
#define TFLOW          12    /* flow control                 */
#define TNODATA        13    /* no data                      */
#define TNODIS         14    /* discon_ind not found on q    */
#define TNOUDERR       15    /* unitdata error not found     */
#define TBADFLAG       16    /* bad flags                    */
#define TNOREL         17    /* no ord rel found on q        */
#define TNOTSUPPORT    18    /* primitive not supported      */
#define TSTATECHNG     19    /* state is changing            */

#define T_LISTEN       0x0001  /* connection indication rcvd */
#define T_CONNECT      0x0002  /* connect confirmation rcvd  */
#define T_DATA         0x0004  /* normal data received       */
#define T_EXDATA       0x0008  /* expedited data received    */
#define T_DISCONNECT   0x0010  /* disconnect received        */
#define T_ERROR        0x0020  /* fatal error occurred       */
#define T_UDERR        0x0040  /* data gram error indication */
#define T_ORDREL       0x0080  /* orderly release indication */
#define T_EVENTS       0x00ff  /* event mask                 */

#define T_MORE         0x001   /* more data            */
#define T_EXPEDITED    0x002   /* expedited data       */
#define T_NEGOTIATE    0x004   /* set opts             */
#define T_CHECK        0x008   /* check opts           */
#define T_DEFAULT      0x010   /* get default opts     */
#define T_SUCCESS      0x020   /* successful           */
#define T_FAILURE      0x040   /* failure              */

struct t_info {
    long addr;      /* size of protocol address      */
    long options;   /* size of protocol options      */
    long tsdu;      /* size of max transport service data unit */
    long etsdu;     /* size of max expedited tsdu  */
    long connect;   /* max data for connection primitives */
    long discon;    /* max data for disconnect primitives */
    long servtype;  /* provider service type              */
};

#define T_COTS     01   /* connection oriented service */
#define T_COTS_ORD 02   /* conn oriented w/orderly release */
#define T_CLTS     03   /* connectionless transport service */
struct netbuf {
        unsigned int maxlen;
        unsigned int len;
        char *buf;
};
```

```
typdef long clock_t;

struct tms {
        clock_t    tms_utime;         /* user time */
        clock_t    tms_stime;         /* system time */
        clock_t    tms_cutime;        /* user time, children */
        clock_t    tms_cstime;        /* system time, children */
};
```

Figure 6-45 <sys/times.h>

```
struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* and microseconds */
};

#define ITIMER_REAL     0
#define ITIMER_VIRTUAL  1
#define ITIMER_PROF     2
#define ITIMER_REALPROF 3

struct  itimerval {
        struct  timeval it_interval;    /* timer interval */
        struct  timeval it_value;       /* current value */
};

#define SEC             1
#define MILLISEC        1000
#define MICROSEC        1000000
#define NANOSEC         1000000000

#define CLOCK_REALTIME  0  /* real (clock on the wall) time */
#define CLOCK_VIRTUAL   1  /* user CPU usage clock */
#define CLOCK_PROF      2  /* user and system CPU usage clock */

#define TIMER_RELTIME   0x0     /* set timer relative */
#define TIMER_ABSTIME   0x1     /* set timer absolute */

typedef struct  timespec {       /* definition per POSIX.4 */
        time_t          tv_sec;  /* seconds */
        long            tv_nsec; /* and nanoseconds */
} timespec_t;

typedef struct itimerspec {         /* definition per POSIX.4 */
        struct timespec it_interval;/* timer period */
        struct timespec it_value;   /* timer expiration */
} itimerspec_t;

struct  tm {
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};

extern long timezone;
extern int daylight;
extern char *tzname[2];
```

Figure 6-44 <sys/time.h>

```
#define NOFLSH  0000200
#define TOSTOP  0000400
#define ECHOCTL 0001000
#define ECHOPRT 0002000
#define ECHOKE  0004000
#define DEFECHO 0010000
#define FLUSHO  0020000
#define PENDIN  0040000
#define IEXTEN  0100000
#define _TIOC   ('T'<<8)

#define TCGETA  (_TIOC|1)
#define TCSETA  (_TIOC|2)
#define TCSETAW (_TIOC|3)
#define TCSETAF (_TIOC|4)
#define TCSBRK  (_TIOC|5)
#define TCXONC  (_TIOC|6)
#define TCFLSH  (_TIOC|7)

#define TCGETS          (_TIOC|13)
#define TCSETS          (_TIOC|14)
#define TCSETSW         (_TIOC|15)
#define TCSETSF         (_TIOC|16)

#define TCIFLUSH        0
#define TCOFLUSH        1
#define TCIOFLUSH       2

#define TCOOFF          0
#define TCOON           1
#define TCIOFF          2
#define TCION           3

#define B0      0
#define B50     1
#define B75     2
#define B110    3
#define B134    4
#define B150    5
#define B200    6
#define B300    7
#define B600    8
#define B1200   9
#define B1800   10
#define B2400   11
#define B4800   12
#define B9600   13
#define B19200  14
#define B38400  15
```

Figure 6-43 `<termios.h>`

```
#define OLCUC   0000002
#define ONLCR   0000004
#define OCRNL   0000010
#define ONOCR   0000020
#define ONLRET  0000040
#define OFILL   0000100
#define OFDEL   0000200
#define NLDLY   0000400
#define NL0     0
#define NL1     0000400
#define CRDLY   0003000
#define CR0     0
#define CR1     0001000
#define CR2     0002000
#define CR3     0003000
#define TABDLY  0014000
#define TAB0    0
#define TAB1    0004000
#define TAB2    0010000
#define TAB3    0014000
#define XTABS   0014000
#define BSDLY   0020000
#define BS0     0
#define BS1     0020000
#define VTDLY   0040000
#define VT0     0
#define VT1     0040000
#define FFDLY   0100000
#define FF0     0
#define FF1     0100000

#define CBAUD   0000017
#define CSIZE   0000060
#define CS5     0
#define CS6     0000020
#define CS7     0000040
#define CS8     0000060
#define CSTOPB  0000100
#define CREAD   0000200
#define PARENB  0000400
#define PARODD  0001000
#define HUPCL   0002000
#define CLOCAL  0004000
#define RCV1EN  0010000
#define XMT1EN  0020000
#define LOBLK   0040000
#define XCLUDE  0100000
#define CRTSCTS 020000000000
#define CIBAUD  03600000
#define PAREXT  04000000

#define ISIG    0000001
#define ICANON  0000002
#define XCASE   0000004
#define ECHO    0000010
#define ECHOE   0000020
#define ECHOK   0000040
#define ECHONL  0000100
```

```
#define NCC     8
#define NCCS    19
#define CTRL(c) ((c)&037)

#define IBSHIFT 16
#define _POSIX_VDISABLE 0

typedef unsigned long tcflag_t;
typedef unsigned char cc_t;
typedef unsigned long speed_t;

/*
 * Ioctl control packet
 */
struct termios {
        tcflag_t    c_iflag;        /* input modes */
        tcflag_t    c_oflag;        /* output modes */
        tcflag_t    c_cflag;        /* control modes */
        tcflag_t    c_lflag;        /* line discipline modes */
        cc_t        c_cc[NCCS];     /* control chars */
};

#define VINTR           0
#define VQUIT           1
#define VERASE          2
#define VKILL           3
#define VEOF            4
#define VEOL            5
#define VEOL2           6
#define VMIN            4
#define VTIME           5
#define VSWTCH          7
#define VSTART          8
#define VSTOP           9
#define VSUSP           10
#define VDSUSP          11
#define VREPRINT        12
#define VDISCARD        13
#define VWERASE         14
#define VLNEXT          15

#define IGNBRK  0000001
#define BRKINT  0000002
#define IGNPAR  0000004
#define PARMRK  0000010
#define INPCK   0000020
#define ISTRIP  0000040
#define INLCR   0000100
#define IGNCR   0000200
#define ICRNL   0000400
#define IUCLC   0001000
#define IXON    0002000
#define IXANY   0004000
#define IXOFF   0010000
#define IMAXBEL 0020000
#define OPOST   0000001
```

```
        int sl_nmods;
        struct str_mlist *sl_modlist;
};

#define ANYMARK          0x01
#define LASTMARK         0x02

struct bandinfo {
        unsigned char   bi_pri;
        int             bi_flag;
};
```

Figure 6-42 `<stropts.h>`

```
#define I_SWROPT        (STR|023)
#define I_GWROPT        (STR|024)
#define I_LIST          (STR|025)
#define I_PLINK         (STR|026)
#define I_PUNLINK       (STR|027)
#define I_SETEV         (STR|030)
#define I_GETEV         (STR|031)
#define I_STREV         (STR|032)
#define I_UNSTREV       (STR|033)
#define I_FLUSHBAND     (STR|034)
#define I_CKBAND        (STR|035)
#define I_GETBAND       (STR|036)
#define I_ATMARK        (STR|037)
#define I_SETCLTIME     (STR|040)
#define I_GETCLTIME     (STR|041)
#define I_CANPUT        (STR|042)

struct strioctl {
        int     ic_cmd;         /* command */
        int     ic_timout;      /* timeout value */
        int     ic_len;         /* length of data */
        char    *ic_dp;         /* pointer to data */
};

#define INFTIM          -1

struct strbuf {
        int     maxlen;         /* no. of bytes in buffer */
        int     len;            /* no. of bytes returned */
        char    *buf;           /* pointer to data */
};

struct strpeek {
        struct strbuf   ctlbuf;
        struct strbuf   databuf;
        long            flags;
};

struct strfdinsert {
        struct strbuf   ctlbuf;
        struct strbuf   databuf;
        long            flags;
        int             fildes;
        int             offset;
};

struct strrecvfd {
        int fd;
        uid_t uid;
        gid_t gid;
        char fill[8];
};

struct str_mlist {
        char l_name[FMNAMESZ+1];
};

struct str_list {
```

```
#define SNDZERO         0x001
#define SNDPIPE         0x002
#define RNORM           0x000
#define RMSGD           0x001
#define RMSGN           0x002
#define RMODEMASK       0x003
#define RPROTDAT        0x004
#define RPROTDIS        0x008
#define RPROTNORM       0x010
#define RPROTMASK       0x01c

#define FLUSHR          0x01
#define FLUSHW          0x02
#define FLUSHRW         0x03
#define FLUSHBAND       0x04

#define S_INPUT         0x0001
#define S_HIPRI         0x0002
#define S_OUTPUT        0x0004
#define S_MSG           0x0008
#define S_ERROR         0x0010
#define S_HANGUP        0x0020
#define S_RDNORM        0x0040
#define S_WRNORM        S_OUTPUT
#define S_RDBAND        0x0080
#define S_WRBAND        0x0100
#define S_BANDURG       0x0200

#define RS_HIPRI        0x01
#define MSG_HIPRI       0x01
#define MSG_ANY         0x02
#define MSG_BAND        0x04

#define MORECTL         1
#define MOREDATA        2

#define MUXID_ALL       (-1)

#define STR             ('S'<<8)
#define I_NREAD         (STR|01)
#define I_PUSH          (STR|02)
#define I_POP           (STR|03)
#define I_LOOK          (STR|04)
#define I_FLUSH         (STR|05)
#define I_SRDOPT        (STR|06)
#define I_GRDOPT        (STR|07)
#define I_STR           (STR|010)
#define I_SETSIG        (STR|011)
#define I_GETSIG        (STR|012)
#define I_FIND          (STR|013)
#define I_LINK          (STR|014)
#define I_UNLINK        (STR|015)
#define I_RECVFD        (STR|016)
#define I_PEEK          (STR|017)
#define I_FDINSERT      (STR|020)
#define I_SENDFD        (STR|021)
```

```
#define BUFSIZ  1024
#define _NFILE  20

#define _IOFBF          0000
#define _IOLBF          0100
#define _IONBF          0004
#define _IOEOF          0020
#define _IOERR          0040

#define EOF             (-1)
#define FOPEN_MAX       _NFILE
#define FILENAME_MAX    1024

#define L_ctermid       9
#define L_cuserid       9

#define P_tmpdir        "/var/tmp/"
#define L_tmpnam        25

#define stdin   (&_ _iob[0])
#define stdout  (&_ _iob[1])
#define stderr  (&_ _iob[2])

typedef struct
{
    int             _cnt;   /* number of avail chars in buf */
    unsigned char   *_ptr;  /* next char from/to here in buf */
    unsigned char   *_base; /* the buffer */
    unsigned char   _flag;  /* the state of the stream */
    unsigned char   _file;  /* UNIX System file descriptor */
} FILE;

extern FILE         _ _iob[_NFILE];
```

Figure 6-40 `<stdio.h>`

```
typedef struct {
        int     quot;
        int     rem;
} div_t;

typedef struct {
        long    quot;
        long    rem;
} ldiv_t;

#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0
#define RAND_MAX        32767

extern unsigned char    _ _ctype[512];
#define MB_CUR_MAX      _ _ctype[520]
```

Figure 6-41 `<stdlib.h>`

```
#define FSTYPSZ 16

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

#define ST_RDONLY   0x01    /* read-only file system */
#define ST_NOSUID   0x02    /* does not support setuid/setgid */
```

Figure 6-38 `<sys/statvfs.h>`

```
#define NULL            0
typedef int             ptrdiff_t;
typedef unsigned int    size_t;
typedef long            wchar_t;
```

Figure 6-39 `<stddef.h>`

```
#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 S_IFMT      0xF000  /* type of file */
#define S_IFIFO     0x1000  /* fifo */
#define S_IFCHR     0x2000  /* character special */
#define S_IFDIR     0x4000  /* directory */
#define S_IFBLK     0x6000  /* block special */
#define S_IFREG     0x8000  /* regular */
#define S_IFLNK     0xA000  /* symbolic link */
#define S_IFSOCK    0xC000  /* socket */
#define S_ISUID     0x800   /* set user id on execution */
#define S_ISGID     0x400   /* set group id on execution */
#define S_ISVTX    0x200   /* save swapped text even after use */
#define S_IREAD     00400   /* read permission, owner */
#define S_IWRITE    00200   /* write permission, owner */
#define S_IEXEC     00100   /* execute/search perm., owner */
#define S_ENFMT     S_ISGID /* record locking enforcement  */
#define S_IRWXU     00700   /* read, write, execute: owner */
#define S_IRUSR     00400   /* read permission: owner */
#define S_IWUSR     00200   /* write permission: owner */
#define S_IXUSR     00100   /* execute permission: owner */
#define S_IRWXG     00070   /* read, write, execute: group */
#define S_IRGRP     00040   /* read permission: group */
#define S_IWGRP     00020   /* write permission: group */
#define S_IXGRP     00010   /* execute permission: group */
#define S_IRWXO     00007   /* read, write, execute: other */
#define S_IROTH     00004   /* read permission: other */
#define S_IWOTH     00002   /* write permission: other */
#define S_IXOTH     00001   /* execute permission: other */
```

Figure 6-37 `<sys/stat.h>`

```
                    clock_t _stime;
                } _cld;
            } _pdata;
        } _proc;

        struct {
            caddr_t _addr;
            int     _trapno;
        } _fault;

        struct {
            int     _fd;
            long    _band;
        } _file;

        struct {
            caddr_t _faddr;
            timestruc_t _tstamp;
            short   _syscall;
            char    _nsysarg;
            char    _fault;
            long    _sysarg[8];
            long    _mstate[17];
        } _prof;
    } _data;
} siginfo_t;
```

Figure 6-36 `<sys/siginfo.h>`

```
#define ILL_ILLOPC      1   /* illegal opcode */
#define ILL_ILLOPN      2   /* illegal operand */
#define ILL_ILLADR      3   /* illegal addressing mode */
#define ILL_ILLTRP      4   /* illegal trap */
#define ILL_PRVOPC      5   /* privileged opcode */
#define ILL_PRVREG      6   /* privileged register */
#define ILL_COPROC      7   /* co-processor */
#define ILL_BADSTK      8   /* bad stack */
#define FPE_INTDIV      1   /* integer divide by zero */
#define FPE_INTOVF      2   /* integer overflow */
#define FPE_FLTDIV      3   /* floating point divide by zero */
#define FPE_FLTOVF      4   /* floating point overflow */
#define FPE_FLTUND      5   /* floating point underflow */
#define FPE_FLTRES      6   /* floating point inexact result */
#define FPE_FLTINV      7   /* invalid floating point op */
#define FPE_FLTSUB      8   /* subscript out of range */
#define SEGV_MAPERR     1   /* address not mapped to object */
#define SEGV_ACCERR     2   /* invalid permissions */
#define BUS_ADRALN      1   /* invalid address alignment */
#define BUS_ADRERR      2   /* non-existent physical address */
#define BUS_OBJERR      3   /* object specific hardware error */
#define TRAP_BRKPT      1   /* breakpoint trap */
#define TRAP_TRACE      2   /* trace trap */
#define CLD_EXITED      1   /* child has exited */
#define CLD_KILLED      2   /* child was killed */
#define CLD_DUMPED      3   /* child has coredumped */
#define CLD_TRAPPED     4   /* traced child has stopped */
#define CLD_STOPPED     5   /* child has stopped on signal */
#define CLD_CONTINUED   6   /* stopped child has continued */
#define POLL_IN         1   /* input available */
#define POLL_OUT        2   /* output possible */
#define POLL_MSG        3   /* message available */
#define POLL_ERR        4   /* I/O error */
#define POLL_PRI        5   /* high priority input available */
#define POLL_HUP        6   /* device disconnected */

#define PROF_SIG        1   /* have to set code nonzero */

#define SI_MAXSZ        128
#define SI_PAD          ((SI_MAXSZ / sizeof (int)) - 3)
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;
                    union sigval    _value;
                } _kill;
                struct {
                    clock_t _utime;
                    int     _status;
```

```
#define SA_SIGINFO   0x00000008

#define SA_NOCLDWAIT 0x00010000 /* don't save zombie children */
#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 6-35 `<signal.h>`

```
#define SIGHUP  1   /* hangup */
#define SIGINT  2   /* interrupt (rubout) */
#define SIGQUIT 3   /* quit (ASCII FS) */
#define SIGILL  4   /* illegal instr. (not reset when caught) */
#define SIGTRAP 5   /* trace trap (not reset when caught) */
#define SIGIOT  6   /* IOT instruction */
#define SIGABRT 6   /* used by abort   */
#define SIGEMT  7   /* EMT instruction */
#define SIGFPE  8   /* floating point exception */
#define SIGKILL 9   /* kill (cannot be caught or ignored) */
#define SIGBUS  10  /* bus error */
#define SIGSEGV 11  /* segmentation violation */
#define SIGSYS  12  /* bad argument to system call */
#define SIGPIPE 13  /* write on a pipe with no one to read it */
#define SIGALRM 14  /* alarm clock */
#define SIGTERM 15  /* software termination signal from kill */
#define SIGUSR1 16  /* user defined signal 1 */
#define SIGUSR2 17  /* user defined signal 2 */
#define SIGCLD  18  /* child status change */
#define SIGCHLD 18  /* child status change alias (POSIX) */
#define SIGPWR  19  /* power-fail restart */
#define SIGWINCH 20 /* window size change */
#define SIGURG  21  /* urgent socket condition */
#define SIGPOLL 22  /* pollable event occured */
#define SIGIO   SIGPOLL /* socket I/O possible (SIGPOLL alias) */
#define SIGSTOP 23  /* stop (cannot be caught or ignored) */
#define SIGTSTP 24  /* user stop requested from tty */
#define SIGCONT 25  /* stopped process has been continued */
#define SIGTTIN 26  /* background tty read attempted */
#define SIGTTOU 27  /* background tty write attempted */
#define SIGVTALRM 28 /* virtual timer expired */
#define SIGPROF 29   /* profiling timer expired */
#define SIGXCPU 30   /* exceeded cpu limit */
#define SIGXFSZ 31   /* exceeded file size limit */
#define SIGWAITING 32   /* process's lwps are blocked */

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

typedef struct {
        unsigned long   _ _sigbits[4];
} sigset_t;

struct sigaction {
        int sa_flags;
        void (*_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 SHM_RDONLY      010000
#define SHM_RND         020000

struct shmid_ds {
    struct ipc_perm shm_perm;   /* operation permission struct */
    int             shm_segsz;  /* size of segment in bytes */
    struct anon_map *shm_amp;   /* segment anon_map pointer */
    unsigned short  shm_lkcnt;  /* number of times locked */
    pid_t           shm_lpid;   /* pid of last shmop */
    pid_t           shm_cpid;   /* pid of creator */
    unsigned long   shm_nattch; /* used only for shminfo */
    unsigned long   shm_cnattch;/* used only for shminfo */
    time_t          shm_atime;  /* last shmat time */
    long            shm_pad1;   /* resv'd for time_t expansion */
    time_t          shm_dtime;  /* last shmdt time */
    long            shm_pad2;   /* resv'd for time_t expansion */
    time_t          shm_ctime;  /* last change time */
    long            shm_pad3;   /* resv'd for time_t expansion */
    long            shm_pad4[4];/* reserve area  */
};
```

Figure 6-34 `<sys/shm.h>`

```
#define SEM_UNDO    010000  /* set up adjust on exit entry */

#define GETNCNT 3       /* get semncnt */
#define GETPID  4       /* get sempid */
#define GETVAL  5       /* get semval */
#define GETALL  6       /* get all semval's */
#define GETZCNT 7       /* get semzcnt */
#define SETVAL  8       /* set semval */
#define SETALL  9       /* set all semval's */

struct semid_ds {
    struct ipc_perm sem_perm;  /* operation permission struct */
    struct sem      *sem_base; /* ptr to first semaphore in set */
    unsigned short  sem_nsems; /* # of semaphores in set */
    time_t          sem_otime; /* last semop time */
    long            sem_pad1;  /* reserved for time_t expansion */
    time_t          sem_ctime; /* last change time */
    long            sem_pad2;  /* time_t expansion */
    long            sem_pad3[4];  /* reserve area */
};

struct sem {
    unsigned short  semval;         /* semaphore value */
    pid_t           sempid;         /* pid of last operation */
    unsigned short  semncnt;        /* # awaiting semval > cval */
    unsigned short  semzcnt;        /* # awaiting semval = 0 */
};

struct sembuf {
    unsigned short  sem_num;        /* semaphore # */
    short           sem_op;         /* semaphore operation */
    short           sem_flg;        /* operation flags */
};
```

Figure 6-32 <sys/sem.h>

```
#define _INT_JBLEN      24
#define _DBL_JBLEN      19
#define _SIGJBLEN       132

typedef struct {
        int int_vals[_INT_JBLEN];
        double dbl_vals[_DBL_JBLEN];
        int pad[2];
} jmp_buf[1];

typedef int sigjmp_buf[_SIGJBLEN];
```

Figure 6-33 <setjmp.h>

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

struct xdr_discrim {
    int        value;
    xdrproc_t proc;
};

enum authdes_namekind {
    ADN_FULLNAME,
    ADN_NICKNAME
};

struct authdes_fullname {
    char *          name;    /* client name, MAXNETNAMELEN max */
    union des_block key;     /* conversation key */
    unsigned long   window;  /* associated window */
};

struct authdes_cred {
    enum authdes_namekind adc_namekind;
    struct authdes_fullname adc_fullname;
    unsigned long adc_nickname;
};

typedef struct XDR {
    enum xdr_op     x_op;    /* operation; fast additional param */
    struct xdr_ops {
        bool_t  (*x_getlong)(struct XDR *, long *);
        bool_t  (*x_putlong)(struct XDR *, long *);
        bool_t  (*x_getbytes)(struct XDR *, caddr_t, int);
        bool_t  (*x_putbytes)(struct XDR *, caddr_t, int);
        u_int   (*x_getpostn)(struct XDR *);
        bool_t  (*x_setpostn)(struct XDR *, u_int);
        long *  (*x_inline)(struct XDR *, int);
        void    (*x_destroy)(struct XDR *);
        bool_t  (*x_control)(struct XDR *, int, void *);
    } *x_ops;
    char *  x_public;    /* users' data */
    char *  x_private;   /* pointer to private data */
    char *  x_base;      /* private used for position info */
    int     x_handy;     /* extra private word */
} XDR;

typedef bool_t (*xdrproc_t)();
```

Figure 6-30 `<rpc.h>`

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

Figure 6-31 `<search.h>`

```
        } AR_versions;
        struct {
            char *  where;
            xdrproc_t proc;
        } AR_results;
        /* and many other null cases */
    } ru;
};

struct rejected_reply {
    enum reject_stat rj_stat;
    union {
        struct {
            unsigned long low;
            unsigned long high;
        } RJ_versions;
        enum auth_stat RJ_why;  /* why auth. did not work */
    } ru;
};

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

struct call_body {
    unsigned long cb_rpcvers;      /* must be equal to two */
    unsigned long cb_prog;
    unsigned long cb_vers;
    unsigned long cb_proc;
    struct opaque_auth cb_cred;
    struct opaque_auth cb_verf;   /*  provided by client */
};

struct rpc_msg {
    unsigned long   rm_xid;
    enum msg_type   rm_direction;
    union {
        struct call_body RM_cmb;
        struct reply_body RM_rmb;
    } ru;
};

struct rpcb {
    unsigned long   r_prog;
    unsigned long   r_vers;
    char *          r_netid;
    char *          r_addr;
    char *          r_owner;
};

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

```
};

typedef struct _ _svcxprt {
    int             xp_fd;
    u_short         xp_port;
    struct  xp_ops  *xp_ops;
    int       xp_addrlen;  /* length of remote addr. Obsoleted */
    char            *xp_tp /* transport provider device name */
    char            *xp_netid;    /* network token */
    struct netbuf   xp_ltaddr;    /* local transport address */
    struct netbuf   xp_rtaddr;    /* remote transport address */
    char       xp_raddr[16]; /* remote address. Now obsoleted */
    struct opaque_auth xp_verf;    /* raw response verifier */
    char *       xp_p1;      /* private: for use by svc ops */
    char *       xp_p2;      /* private: for use by svc ops */
    char *       xp_p3;      /* private: for use by svc lib */
} SVCXPRT;

struct svc_req {
    u_long            rq_prog;      /* service program number */
    u_long            rq_vers;      /* service protocol version */
    u_long            rq_proc;      /* the desired procedure */
    struct opaque_auth rq_cred;     /* raw creds from the wire */
    caddr_t           rq_clntcred;  /* read only cooked cred */
    struct _ _svcxprt  *rq_xprt;      /* associated transport */
};

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
};

struct accepted_reply {
    struct opaque_auth      ar_verf;
    enum accept_stat        ar_stat;
    union {
        struct {
                unsigned long   low;
                unsigned long   high;
```

```
            u_long high;    /* highest version supported */
        } RE_vers;
        struct {      /* maybe meaningful if RPC_FAILED */
            long s1;
            long s2;
        } RE_lb;      /* life boot & debugging only */
    } ru;
};

struct rpc_createerr {
    enum clnt_stat cf_stat;
    struct rpc_err cf_error; /* useful when RPC_PMAPFAILURE */
};

typedef struct _ _client {
    AUTH    *cl_auth;        /* authenticator */
    struct clnt_ops {
        enum clnt_stat  (*cl_call)(struct _ _client *, u_long,
                           xdrproc_t, caddr_t, xdrproc_t,
                           caddr_t, struct timeval);
        void            (*cl_abort)();  /* abort a call */
        void            (*cl_geterr)(struct _ _client *,
                                     struct rpc_err *);
        bool_t          (*cl_freeres)(struct _ _client *,
                                        xdrproc_t, caddr_t);
        void            (*cl_destroy)(struct _ _client *);
        bool_t          (*cl_control)(struct _ _client *, int,
                                        char *);
        int             (*cl_settimers)(struct _ _client *,
                            struct rpc_timers *,
                            struct rpc_timers *, int,
                            void (*)(), caddr_t, u_long);
    } *cl_ops;
    char *                  cl_private;     /* private stuff */
    char                    *cl_netid;      /* network token */
    char                    *cl_tp;         /* device name */
} CLIENT;

#define FEEDBACK_REXMIT1   1  /* first retransmit */
#define FEEDBACK_OK        2  /* no retransmits */

#define CLSET_TIMEOUT       1 /* set timeout (timeval) */
#define CLGET_TIMEOUT       2 /* get timeout (timeval) */
#define CLGET_SERVER_ADDR   3 /* get server's (sockaddr) */
#define CLGET_FD            6 /* get connections file descr */
#define CLGET_SVC_ADDR      7 /* get server's addr (netbuf) */
#define CLSET_FD_CLOSE      8 /* close fd while clnt_destroy */
#define CLSET_FD_NCLOSE     9 /* Do not close fd while */
                              /* clnt_destroy          */
#define CLSET_RETRY_TIMEOUT 4 /* set retry timeout (timeval) */
#define CLGET_RETRY_TIMEOUT 5 /* get retry timeout (timeval) */

extern struct rpc_createerr rpc_createerr;

enum xprt_stat {
    XPRT_DIED,
    XPRT_MOREREQS,
    XPRT_IDLE
```

```
#define AUTH_NULL   0          /* backward compatibility */
#define AUTH_SYS    1          /* unix style (uid, gids) */
#define AUTH_UNIX   AUTH_SYS
#define AUTH_SHORT  2     /* short hand unix style  */
#define AUTH_DES    3    /* des style (encrypted timestamps) */

enum clnt_stat {
    RPC_SUCCESS = 0,           /* call succeeded */
    RPC_CANTENCODEARGS = 1,    /* can't encode arguments */
    RPC_CANTDECODERES = 2,     /* can't decode results */
    RPC_CANTSEND = 3,          /* failure in sending call */
    RPC_CANTRECV = 4,          /* failure in receiving result */
    RPC_TIMEDOUT = 5,          /* call timed out */
    RPC_INTR = 18,             /* call interrupted */
    RPC_UDERROR = 23,          /* recv got uderr indication */
    RPC_VERSMISMATCH = 6,      /* rp versions not compatible */
    RPC_AUTHERROR = 7,         /* authentication error */
    RPC_PROGUNAVAIL = 8,       /* program not available */
    RPC_PROGVERSMISMATCH = 9,  /* program version mismatched */
    RPC_PROCUNAVAIL = 10,      /* procedure unavailable */
    RPC_CANTDECODEARGS = 11,   /* decode arguments error */
    RPC_SYSTEMERROR = 12,      /* generic "other problem" */
    RPC_UNKNOWNHOST = 13,      /* unknown host name */
    RPC_UNKNOWNPROTO = 17,     /* unknown protocol */
    RPC_UNKNOWNADDR = 19,      /* Remote address unknown */
    RPC_NOBROADCAST = 21,      /* Broadcasting not supported */
    RPC_RPCBFAILURE = 14,  /* the pmapper failed in its call */
    RPC_PROGNOTREGISTERED = 15,/* remote prog not registered */
    RPC_N2AXLATEFAILURE = 22,  /* name to address translation
                               /*    failed*/
    RPC_TLIERROR = 20,         /* misc error in TLI library */
    RPC_FAILED = 16,           /* unspecified error */
};
#define RPC_PMAPFAILURE RPC_RPCBFAILURE

#define _RPC_NONE       0
#define _RPC_NETPATH    1
#define _RPC_VISIBLE    2
#define _RPC_CIRCUIT_V  3
#define _RPC_DATAGRAM_V 4
#define _RPC_CIRCUIT_N  5
#define _RPC_DATAGRAM_N 6
#define _RPC_TCP        7
#define _RPC_UDP        8

#define RPC_ANYSOCK     -1
#define RPC_ANYFD       RPC_ANYSOCK

struct rpc_err {
    enum clnt_stat re_status;
    union {
        struct {
            int RE_errno;   /* related system error */
            int RE_t_errno; /* related tli error number */
        } RE_err;
        enum auth_stat RE_why;  /* why auth error occurred */
        struct {
           u_long low;     /* lowest version supported */
```

```
#define MAX_AUTH_BYTES  400
#define MAXNETNAMELEN   255    /* max length of net user's name */
#define HEXKEYBYTES 48

enum auth_stat {
    AUTH_OK = 0,
    AUTH_BADCRED = 1,      /* bogus credentials (seal broken) */
    AUTH_REJECTEDCRED = 2, /* client should begin new session */
    AUTH_BADVERF = 3,      /* bogus verifier (seal broken) */
    AUTH_REJECTEDVERF = 4,  /* verifier expired or replayed */
    AUTH_TOOWEAK = 5,      /* rejected due to security reasons */
    AUTH_INVALIDRESP = 6,  /* bogus response verifier */
    AUTH_FAILED = 7,       /* some unknown reason */
};

union des_block {
    struct  {
            unsigned long high;
            unsigned long low;
    } key;
    char c[8];
};

struct opaque_auth {
    int           oa_flavor;    /* flavor of auth */
    char *        oa_base;      /* address of more auth stuff */
    unsigned int  oa_length;    /* not to exceed MAX_AUTH_BYTES */
};

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

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;
};

extern struct opaque_auth _null_auth;

#define AUTH_NONE   0           /* no authentication */
```

```
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 6-28 `<pwd.h>`

```
#define RLIMIT_CPU      0         /* cpu time in milliseconds */
#define RLIMIT_FSIZE    1         /* maximum file size */
#define RLIMIT_DATA     2         /* data size */
#define RLIMIT_STACK    3         /* stack size */
#define RLIMIT_CORE     4         /* core file size */
#define RLIMIT_NOFILE   5         /* file descriptors */
#define RLIMIT_VMEM     6         /* maximum mapped memory */
#define RLIMIT_AS       RLIMIT_VMEM

typedef unsigned long rlim_t;

struct rlimit {
        rlim_t  rlim_cur;         /* current limit */
        rlim_t  rlim_max;         /* maximum value for rlim_cur */
};
```

Figure 6-29 `<sys/resource.h>`

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

typedef long id_t;

typedef enum idtype {
    P_PID,          /* A process identifier.                 */
    P_PPID,         /* A parent process identifier.          */
    P_PGID,         /* A process group (job control group)   */
                    /* identifier.                           */
    P_SID,          /* A session identifier.                 */
    P_CID,          /* A scheduling class identifier.        */
    P_UID,          /* A user identifier.                    */
    P_GID,          /* A group identifier.                   */
    P_ALL,          /* All processes.                        */
    P_LWPID         /* An LWP identifier.                    */
} idtype_t;

typedef enum idop {
    POP_DIFF,       /* Set difference.  The processes which */
                    /* are in the left operand set and not  */
                    /* in the right operand set.            */
    POP_AND,        /* Set disjunction.  The processes      */
                    /* which are in both the left and right */
                    /* operand sets.                        */
    POP_OR,         /* Set conjunction.  The processes      */
                    /* which are in either the left or the  */
                    /* right operand sets (or both).        */
    POP_XOR         /* Set exclusive or.  The processes     */
                    /* which are in either the left or      */
                    /* right operand sets but not in both.  */
} idop_t;

typedef struct procset {
    idop_t          p_op;   /* The operator connection      */
                            /* between the following two    */
                            /* operands each of which is a  */
                            /* simple set of processes.     */

    idtype_t        p_lidtype;
                            /* The type of the left operand */
                            /* simple set.                  */
    id_t            p_lid;  /* The id of the left operand.  */

    idtype_t        p_ridtype;
                            /* The type of the right        */
                            /* operand simple set.          */
    id_t            p_rid;  /* The id of the right operand. */
} procset_t;

#define P_MYID  (-1)
```

Figure 6-27 `<sys/procset.h>`

```
#define CANBSIZ        256     /* max size of typewriter line  */
#define NGROUPS_UMIN    0
#define MAXPATHLEN      1024
#define MAXSYMLINKS     20
#define MAXNAMELEN      256
#define NADDR           13
#define PIPE_MAX        5120
```

Figure 6-25 `<sys/param.h>`

```
typedef struct pollfd {
        int fd;                  /* file desc to poll */
        short events;            /* events of interest on fd */
        short revents;           /* events that occurred on fd */
} pollfd_t;

#define POLLIN     0x0001   /* fd is readable */
#define POLLPRI    0x0002   /* high priority info at fd */
#define POLLOUT    0x0004   /* fd is writable (won't block) */
#define POLLRDNORM 0x0040   /* normal data is readable */
#define POLLWRNORM POLLOUT
#define POLLRDBAND 0x0080   /* out-of-band data is readable */
#define POLLWRBAND 0x0100   /* out-of-band data is writable */
#define POLLNORM   POLLRDNORM

#define POLLERR    0x0008   /* fd has error condition */
#define POLLHUP    0x0010   /* fd has been hung up on */
#define POLLNVAL   0x0020   /* invalid pollfd entry */
```

Figure 6-26 `<poll.h>`

```
struct nd_addrlist {
        int             n_cnt;          /* number of netbufs */
        struct netbuf   *n_addrs;       /* the netbufs */
};

struct nd_hostservlist {
       int                  h_cnt;   /* number of nd_hostservs */
        struct nd_hostserv     *h_hostservs;   /* the entries */
};

struct nd_hostserv {
        char            *h_host;        /* the host name */
        char            *h_serv;        /* the service name */
};

#define ND_BADARG       -2    /* Bad arguments passed         */
#define ND_NOMEM        -1    /* No virtual memory left       */
#define ND_OK           0     /* Translation successful       */
#define ND_NOHOST       1     /* Hostname was not resolvable  */
#define ND_NOSERV       2     /* Service was unknown          */
#define ND_NOSYM        3     /* Couldn't resolve symbol      */
#define ND_OPEN         4     /* File couldn't be opened      */
#define ND_ACCESS       5     /* File is not accessible       */
#define ND_UKNWN        6     /* Unknown object to be freed   */
#define ND_NOCTRL       7     /* Unknown netdir_options option*/
#define ND_FAILCTRL     8     /* Opt failed in netdir_options */
#define ND_SYSTEM       9     /* Other System error           */

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

#define ND_SET_BROADCAST      1 /* Do t_optmgmt for broadcast */
#define ND_SET_RESERVEDPORT   2 /* bind it to reserve address */
#define ND_CHECK_RESERVEDPORT 3 /* check if address is resv'd */
#define ND_MERGEADDR          4 /* Merge universal address    */

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

Figure 6-23 <netdir.h>

```
#define NL_SETD      1   /* XPG3 Conformant Default set number.*/
#define NL_CAT_LOCALE  (-1) /* XPG4 requirement */

typedef int nl_item;   /* XPG3 Conformant for nl_langinfo(). */
```

Figure 6-24 <nl_types.h>

```
struct  netconfig {
    char            *nc_netid;      /* network identifier  */
    unsigned long   nc_semantics;   /* defined below       */
    unsigned long   nc_flag;        /* defined below       */
    char            *nc_protofmly;  /* protocol family name*/
    char            *nc_proto;      /* protocol name       */
    char            *nc_device;     /* device name for net id */
    unsigned long    nc_nlookups;/* # of ents in nc_lookups   */
   char          **nc_lookups;  /* list of lookup directories */
    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
#define NC_BROADCAST    02
#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_NOPROTO      "-"
#define NC_TCP          "tcp"
#define NC_UDP          "udp"
#define NC_ICMP         "icmp"
```

Figure 6-22 `<netconfig.h>`

```
struct msqid_ds {
    struct ipc_perm msg_perm;   /* operation perm struct */
    struct msg      *msg_first; /* ptr to first message on q */
    struct msg      *msg_last;  /* ptr to last message on q */
    unsigned long   msg_cbytes; /* current # bytes on q */
    unsigned long   msg_qnum;   /* # of messages on q */
    unsigned long   msg_qbytes; /* max # of bytes on q */
    pid_t           msg_lspid;  /* pid of last msgsnd */
    pid_t           msg_lrpid;  /* pid of last msgrcv */
    time_t          msg_stime;  /* last msgsnd time */
    long            msg_pad1;   /* resv'd for time_t expansion */
    time_t          msg_rtime;  /* last msgrcv time */
    long            msg_pad2;   /* time_t expansion */
    time_t          msg_ctime;  /* last change time */
    long            msg_pad3;   /* time expansion */
    long            msg_pad4[4];/* reserve area */
};

#define MSG_NOERROR     010000  /* no error if big message */
```

Figure 6-21 <sys/msg.h>

```
typedef union _h_val {
        unsigned long _i[2];
        double _d;
} _h_val;

extern const _h_val __huge_val;
```

Figure 6-18 <math.h>

```
#define PROT_NONE       0x0          /* pages can't be accessed */
#define PROT_READ       0x1          /* pages can be read */
#define PROT_WRITE      0x2          /* pages can be written */
#define PROT_EXEC       0x4          /* pages can be executed */

#define MAP_SHARED      1            /* share changes */
#define MAP_PRIVATE     2            /* changes are private */
#define MAP_FIXED       0x10         /* user assigns address */

#define MS_SYNC         0x0          /* wait for msync */
#define MS_ASYNC        0x1          /* return immediately */
#define MS_INVALIDATE   0x2          /* invalidate caches */
```

Figure 6-19 <sys/mman.h>

```
#define MS_RDONLY       0x01    /* Read-only */
#define MS_DATA         0x04    /* 6-argument mount */
#define MS_NOSUID       0x10    /* Setuid programs disallowed */
#define MS_REMOUNT      0x20    /* Remount */
```

Figure 6-20 <sys/mount.h>

```
#define ERA_D_FMT        52  /* era date format string */
#define ERA_D_T_FMT      53  /* era date and time format */
#define ERA_T_FMT        54  /* era time format string */
#define ALT_DIGITS       55  /* alternate symbols for digits */
#define YESEXPR          56  /* affirmative response expr. */
#define NOEXPR           57  /* negative response expression */
```

Figure 6-15 `<langinfo.h>`

```
#define MB_LEN_MAX       5
#define TMP_MAX          17576   /* 26 * 26 * 26 */
#define NL_ARGMAX        9       /* max value of "digit" */
#define NL_LANGMAX       14      /* max bytes in a LANG name */
#define NL_MSGMAX        32767   /* max message number */
#define NL_NMAX        1    /* max bytes in N-to-1 mapping chars */
#define NL_SETMAX        255     /* max set number */
#define NL_TEXTMAX       2048    /* max set number */
#define NZERO            20      /* default process priority */
#define FCHR_MAX        1048576 /* max size of a file in bytes */
```

Figure 6-16 `<limits.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 6-17 `<locale.h>`

```
#define DAY_1      1  /* sunday */
#define DAY_2      2  /* monday */
#define DAY_3      3  /* tuesday */
#define DAY_4      4  /* wednesday */
#define DAY_5      5  /* thursday */
#define DAY_6      6  /* friday */
#define DAY_7      7  /* saturday */

#define ABDAY_1    8  /* sun */
#define ABDAY_2    9  /* mon */
#define ABDAY_3    10 /* tue */
#define ABDAY_4    11 /* wed */
#define ABDAY_5    12 /* thu */
#define ABDAY_6    13 /* fri */
#define ABDAY_7    14 /* sat */

#define MON_1      15 /* january */
#define MON_2      16 /* february */
#define MON_3      17 /* march */
#define MON_4      18 /* april */
#define MON_5      19 /* may */
#define MON_6      20 /* june */
#define MON_7      21 /* july */
#define MON_8      22 /* august */
#define MON_9      23 /* september */
#define MON_10     24 /* october */
#define MON_11     25 /* november */
#define MON_12     26 /* december */

#define ABMON_1    27 /* jan */
#define ABMON_2    28 /* feb */
#define ABMON_3    29 /* mar */
#define ABMON_4    30 /* apr */
#define ABMON_5    31 /* may */
#define ABMON_6    32 /* jun */
#define ABMON_7    33 /* jul */
#define ABMON_8    34 /* aug */
#define ABMON_9    35 /* sep */
#define ABMON_10   36 /* oct */
#define ABMON_11   37 /* nov */
#define ABMON_12   38 /* dec */

#define RADIXCHAR 39 /* radix character */
#define THOUSEP   40 /* separator for thousand */
#define YESSTR    41 /* affirmative response for queries */
#define NOSTR     42 /* negative response for queries */
#define CRNCYSTR  43 /* currency symbol */

#define D_T_FMT   44 /* string for formatting date and time */
#define D_FMT     45 /* date format */
#define T_FMT     46 /* time format */
#define AM_STR    47 /* am string */
#define PM_STR    48 /* pm string */

#define CODESET         49   /* codeset name */
#define T_FMT_AMPM      50   /* am or pm time format string */
#define ERA             51  /* era description segments */
```

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

Figure 6-13 <grp.h>

```
struct ipc_perm {
        uid_t           uid;    /* owner's user id */
        gid_t           gid;    /* owner's group id */
        uid_t           cuid;   /* creator's user id */
        gid_t           cgid;   /* creator's group id */
        mode_t          mode;   /* access modes */
        unsigned long   seq;    /* slot usage sequence number */
        key_t           key;    /* key */
        long            pad[4]; /* reserve area */
};

#define IPC_CREAT    0001000  /* create if key doesn't exist */
#define IPC_EXCL     0002000  /* fail if key exists */
#define IPC_NOWAIT   0004000  /* error if request must wait */

#define IPC_PRIVATE     (key_t)0   /* private key */

#define IPC_RMID        10      /* remove identifier */
#define IPC_SET         11      /* set options */
#define IPC_STAT        12      /* get options */
```

Figure 6-14 <sys/ipc.h>

```
#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

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

#define MM_NULLLBL      ((char *) 0)
#define MM_NULLSEV      MM_NOSEV
#define MM_NULLMC       0L
#define MM_NULLTXT      ((char *) 0)
#define MM_NULLACT      ((char *) 0)
#define MM_NULLTAG      ((char *) 0)

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

Figure 6-11 `<fmtmsg.h>`

```
#define FTW_F   0       /* file */
#define FTW_D   1       /* directory */
#define FTW_DNR 2       /* directory without read permission */
#define FTW_NS  3       /* unknown type, stat failed */
#define FTW_SL  4       /* symbolic link */
#define FTW_DP  6       /* directory */

#define FTW_PHYS        01  /* use lstat instead of stat */
#define FTW_MOUNT       02  /* do not cross a mount point */
#define FTW_CHDIR       04  /* chdir to each directory before /*
                               /*reading */
#define FTW_DEPTH      010 /* call descendents before calling */
                             /* the parent */

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

Figure 6-12 `<ftw.h>`

```
#define O_RDONLY    0     /* read only */
#define O_WRONLY    1     /* write only */
#define O_RDWR      2     /* read and write */
#define O_APPEND    0x08  /* append (writes guaranteed at end)*/
#define O_SYNC      0x10  /* synchronized file update option */
#define O_NONBLOCK  0x80  /* non-blocking I/O (POSIX) */
#define O_CREAT     0x100 /* open with file create */
#define O_TRUNC     0x200 /* open with truncation */
#define O_EXCL      0x400 /* exclusive open */
#define O_NOCTTY    0x800 /* don't allocate controlling tty */

#define F_DUPFD         0       /* Duplicate fildes */
#define F_GETFD         1       /* Get fildes flags */
#define F_SETFD         2       /* Set fildes flags */
#define F_GETFL         3       /* Get file flags */
#define F_SETFL         4       /* Set file flags */
#define F_SETLK         6       /* Set file lock */
#define F_SETLKW        7       /* Set file lock and wait */
#define F_GETLK         14      /* Get file lock */

/*
 * File segment locking set data type
 * Information passed to system by user.
 */
typedef struct flock {
        short   l_type;
        short   l_whence;
        off_t   l_start;
        off_t   l_len;     /* len == 0 means until end of file */
        long    l_sysid;
        pid_t   l_pid;
        long    l_pad[4];               /* reserve area */
} flock_t;

/*
 * File segment locking types.
 */
#define F_RDLCK         01      /* Read lock */
#define F_WRLCK         02      /* Write lock */
#define F_UNLCK         03      /* Remove lock(s) */

/*
 * POSIX constants
 */

#define O_ACCMODE       3       /* Mask for file access modes */
#define FD_CLOEXEC      1       /* close on exec flag */
```

Figure 6-9 `<fcntl.h>`

```
extern int __flt_rounds;
```

Figure 6-10 `<float.h>`

```
#define EBADSLT 55  /* invalid slot */
#define EDEADLOCK 56 /* file locking deadlock error */
#define EBFONT  57  /* bad font file fmt */
#define ENOSTR  60  /* Device not a stream */
#define ENODATA 61  /* no data (for no delay io) */
#define ETIME   62  /* timer expired */
#define ENOSR   63  /* out of streams resources */
#define ENONET  64  /* Machine is not on the network */
#define ENOPKG  65  /* Package not installed */
#define EREMOTE 66   /* The object is remote */
#define ENOLINK 67   /* the link has been severed */
#define EADV    68   /* advertise error */
#define ESRMNT  69   /* srmount error */
#define ECOMM   70   /* Communication error on send */
#define EPROTO  71   /* Protocol error */
#define EMULTIHOP 74 /* multihop attempted */
#define EBADMSG 77   /* trying to read unreadable message */
#define ENAMETOOLONG 78 /* path name is too long */
#define EOVERFLOW 79 /* value too large for data type */
#define ENOTUNIQ 80  /* given log. name not unique */
#define EBADFD  81   /* f.d. invalid for this operation */
#define EREMCHG 82   /* Remote address changed */
#define ELIBACC 83   /* Can't access a needed shared lib.*/
#define ELIBBAD 84   /* Accessing a corrupted shared lib.*/
#define ELIBSCN 85   /* .lib section in a.out corrupted.*/
#define ELIBMAX 86   /* Attempting to link in too many libs */
#define ELIBEXEC 87  /* Attempting to exec a shared library */
#define EILSEQ  88   /* Illegal byte sequence */
#define ENOSYS  89   /* Unsupported file system operation */
#define ELOOP   90   /* Symbolic link loop */
#define ERESTART 91  /* Restartable system call */
#define ESTRPIPE 92  /* pipe/FIFO: no sleep in stream head */
#define ENOTEMPTY 93 /* directory not empty */
#define EUSERS  94   /* Too many users (for UFS) */
#define ESTALE 151   /* Stale NFS file handle */
```

Figure 6-8 `<errno.h>`

```
extern int errno;

#define EPERM   1   /* Not super-user */
#define ENOENT  2   /* No such file or directory */
#define ESRCH   3   /* No such process */
#define EINTR   4   /* interrupted system call */
#define EIO     5   /* I/O error */
#define ENXIO   6   /* No such device or address */
#define E2BIG   7   /* Arg list too long */
#define ENOEXEC 8   /* Exec format error */
#define EBADF   9   /* Bad file number */
#define ECHILD  10  /* No children */
#define EAGAIN  11  /* Resource temporarily unavailable */
#define ENOMEM  12  /* Not enough core */
#define EACCES  13  /* Permission denied */
#define EFAULT  14  /* Bad address */
#define ENOTBLK 15  /* Block device required */
#define EBUSY   16  /* Mount device busy */
#define EEXIST  17  /* File exists */
#define EXDEV   18  /* Cross-device link */
#define ENODEV  19  /* No such device */
#define ENOTDIR 20  /* Not a directory */
#define EISDIR  21  /* Is a directory */
#define EINVAL  22  /* Invalid argument */
#define ENFILE  23  /* File table overflow */
#define EMFILE  24  /* Too many open files */
#define ENOTTY  25  /* Inappropriate ioctl for device */
#define ETXTBSY 26  /* Text file busy */
#define EFBIG   27  /* File too large */
#define ENOSPC  28  /* No space left on device */
#define ESPIPE  29  /* Illegal seek */
#define EROFS   30  /* Read only file system */
#define EMLINK  31  /* Too many links */
#define EPIPE   32  /* Broken pipe */
#define EDOM    33  /* Math arg out of domain of func */
#define ERANGE  34  /* Math result not representable */
#define ENOMSG  35  /* No message of desired type */
#define EIDRM   36  /* Identifier removed */
#define ECHRNG  37  /* Channel number out of range */
#define EL2NSYNC 38 /* Level 2 not synchronized */
#define EL3HLT  39  /* Level 3 halted */
#define EL3RST  40  /* Level 3 reset */
#define ELNRNG  41  /* Link number out of range */
#define EUNATCH 42  /* Protocol driver not attached */
#define ENOCSI  43  /* No CSI structure available */
#define EL2HLT  44  /* Level 2 halted */
#define EDEADLK 45  /* Deadlock condition.*/
#define ENOLCK  46  /* No record locks available.*/
#define ECANCELED 47 /* Operation canceled */
#define ENOTSUP 48  /* Operation not supported */
#define EBADE   50  /* invalid exchange */
#define EBADR   51  /* invalid request descriptor */
#define EXFULL  52  /* exchange full */
#define ENOANO  53  /* no anode */
#define EBADRQC 54  /* invalid request code */
```

System Data Interfaces

Data Definitions

This section contains standard header files that describe system data. These files are referred to by the names in angle backets: <*name*.h> and <sys/*name*.h>. Included in these headers 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 on which it will run. While an ABI-conforming system must provide these interfaces, the system does not have to include the actual header files referenced here.

Programmers should observe that the source of the structures defined in these headers is defined in the *System V Interface Definition*.

ANSI C serves as the ABI reference programming language, and data definitions are specified 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.

```
#define _U      01      /* Upper case */
#define _L      02      /* Lower case */
#define _N      04      /* Numeral (digit) */
#define _S      010     /* Spacing character */
#define _P      020     /* Punctuation */
#define _C      040     /* Control character */
#define _B      0100    /* Blank */
#define _X      0200    /* heXadecimal digit */

extern unsigned char    __ctype[521];
```

Figure 6-6 <ctype.h>

```
struct dirent {
       ino_t        d_ino;    /* "inode number" of entry */
       off_t        d_off;    /* offset of disk directory entry */
       unsigned short  d_reclen;    /* length of this record */
       char            d_name[1];   /* name of file */
};
```

Figure 6-7 <dirent.h>

`long long wstoll( const wchar_t *a, wchar_t **b, int c )`

This function converts the base-c string pointed to by `a` to a `signed long long` value and returns this value. If `b` is non-NULL, `*b` is set to point to the first wide-character of `a` that is not interpreted as part of the converted value.

Global Data Symbols

The `libsys` library requires that some global 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 symbol must be provided in the system library on all ABI-conforming systems implemented with the PowerPC Architecture. Declarations for the data objects listed below can be found in the **Data Definitions** section of this chapter.

`__huge_val`

Figure 6-5 `libsys` Global External Data Symbols

Application Constraints

As described above, `libsys` 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 PowerPC Architecture are also required to provide the following symbols:

`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 nonzero, and the program wants ANSI C semantics.

Optional Routines

Note that the facilities and interfaces described in this section are optional components of the PowerPC Processor ABI Supplement.

In addition to the routines specified in the System V ABI, `libc` may also contain the following routines:

atoll	llabs	lldiv	lltostr
strtoll	strtoull	ulltostr	wstoll

Figure 6-4 `libc` Optional Routines

The following routines are 64-bit counterparts to 32-bit routines specified in the System V ABI. It is beneficial if implementations that provide 64-bit signed and unsigned integer data types include these routines in `libc`. In the descriptions below, the non-standard C names `long long` (or `signed long long`) and `unsigned long long` are used to refer to these types. The routines employ the standard calling sequence described in **Function Calling Sequence** in Chapter 3; each `long long` argument and return value is treated in the same manner as a structure consisting solely of two `long`s. Descriptions are written from the caller's point of view with respect to register usage and stack frame layout.

`long long atoll( const char *a )`
This function converts the decimal string pointed to by `a` to a `signed long long` value and returns this value.

`long long llabs( long long a )`
This function returns the absolute value of `a`.

`lldiv_t lldiv( long long a, long long b )`
This function divides `a` by `b` and returns a structure (`lldiv_t`) containing the `long long` quotient and remainder. This structure is the same as the `div_t` structure described in the *System V Interface Definition*, except that the `quot` and `rem` members are of type `long long` instead of `int`.

`char *lltostr( long long a, char *b )`
This function returns a pointer to the string represented by the `long long` value `a`.

`long long strtoll( const char *a, char **b, int c )`
This function converts the base-c string pointed to by `a` to a `signed long long` value and returns this value. If `b` is non-NULL, `*b` is set to point to the first character of `a` that is not interpreted as part of the converted value.

`unsigned long long strtoull( const char *a, char **b, int c )`
This function converts the base-c string pointed to by `a` to an `unsigned long long` value and returns this value. If `b` is non-NULL, `*b` is set to point to the first character of `a` that is not interpreted as part of the converted value.

`char *ulltostr( unsigned long long a, char *b )`
This function returns a pointer to the string represented by the `unsigned long long` value `a`.

0 - `arg_ARGPOINTER`

A `struct`, `union`, or `long double` argument represented in the PowerPC calling conventions as a pointer to (a copy of) the object.

1 - `arg_WORD`

A 32-bit aligned word argument, any of the simple integer types, or a pointer to an object of any type.

2 - `arg_DOUBLEWORD`

A `long long` argument.

3 - `arg_ARGREAL`

A `double` argument. Note that `float` arguments are converted to and passed as `double` arguments.

The mechanism for locating tags, described in **Locating Tags** in Chapter 4, involves the following three functions:

`__add_module_tags(struct module_tags *mt)`

This function adds the tag section described by the `mt` argument to the list of active tag sections.

`__delete_module_tags(struct module_tags *mt)`

This function removes the tag section described by the `mt` argument from the list of active tag sections.

`__tag_lookup_pc(caddr_t pc)`

This function returns a pointer to the `module_tags` structure that describes the tags section applicable to the given PC value, or NULL if there is no applicable tags section.

C Library

Required Routines

An implementation must provide the following processor-specific support routines in `libc`.

`__va_arg`	`__tag_register`	`__tag_deregister`	`__tag_lookup`

Figure 6-3 `libc` Required Routines

`void * __va_arg(va_list argp, _va_arg_type type)`

This function is used by the `va_arg` macros of `<stdarg.h>` and `<varargs.h>`, and it returns a pointer to the next argument specified in the variable argument list `argp`. A variable argument list is an array of one structure, as shown below.

```
void * __va_arg(va_list argp, _va_arg_type type)

/* overflow_arg_area is initially the address at which the
 * first arg passed on the stack, if any, was stored.
 *
 * reg_save_area is the start of where r3:r10 were stored.
 * reg_save_area must be a doubleword aligned.
 *
 * If f1:f8 have been stored (because CR bit 6 was 1),
 * reg_save_area+4*8 must be the start of where f1:f8
 * were stored
 */

typedef struct {
    char gpr;      /* index into the array of 8 GPRs
                    * stored in the register save area
                    * gpr=0 corresponds to r3,
                    * gpr=1 to r4, etc.
                    */
    char fpr;      /* index into the array of 8 FPRs
                    * stored in the register save area
                    * fpr=0 corresponds to f1,
                    * fpr=1 to f2, etc.
                    */
    char *overflow_arg_area;
                   /* location on stack that holds
                    * the next overflow argument
                    */
    char *reg_save_area;
                  /* where r3:r10 and f1:f8 (if saved)
                   * are stored
                   */
} va_list[1];
```

The argument is assumed to be of type `type`. The types are:

`long long __div64( long long a, long long b )`

This function computes the quotient `a / b`, truncating any fractional part, and returns the `signed long long` result.

`long long __dtoll( double a )`

This function converts the double precision value of `a` to a `signed long long` by truncating any fractional part and returns the `signed long long` value.

`unsigned long long __dtoull( double a )`

This function converts the double precision value of `a` to an `unsigned long long` by truncating any fractional part and returns the `unsigned long long` value.

`long long __rem64( long long a, long long b )`

This function computes the remainder upon dividing `a` by `b` and returns the `signed long long` result.

`unsigned long long __udiv64( unsigned long long a, unsigned long long b )`

This function computes the quotient `a / b`, truncating any fractional part, and returns the `unsigned long long` result.

`unsigned long long __urem64( unsigned long long a, unsigned long long b )`

This function computes the remainder upon dividing `a` by `b` and returns the `unsigned long long` result.

Optional Support Routines

Note that the facilities and interfaces described in this section are optional components of the PowerPC Processor ABI Supplement.

In addition to the processor-specific routines specified above, `libsys` may also contain the following processor-specific support routines.

`_q_lltoq`	`_q_qtoll`	`_q_qtoull`	`_q_ulltoq`
`__div64`	`__dtoll`	`__dtoull`	`__rem64`
`__udiv64`	`__urem64`		

Figure 6-2 `libsys` Optional Support Routines

The following routines support software emulation of arithmetic operations for implementations that provide 64-bit signed and unsigned integer data types. In the descriptions below, the non-standard C names `long long` (or `signed long long`) and `unsigned long long` are used to refer to these types. The routines employ the standard calling sequence described in **Function Calling Sequence** in Chapter 3. Descriptions are written from the caller's point of view with respect to register usage and stack frame layout.

Note that the functions prefixed by `_q_` below implement extended precision arithmetic operations. The following restriction applies to each of these functions:

> If any floating-point exceptions occur, the appropriate exception bits in the `FPSCR` are updated; if the corresponding exception is enabled, the floating-point exception trap handler is invoked.

Note – The references in the following descriptions to `a` and `b`, where the corresponding arguments are pointers to `long double` quantities, refer to the values pointed to, not the pointers themselves.

`long double _q_lltoq( long long a )`
: This function converts the `long long` value of `a` to extended precision and returns the extended precision value.

`long long _q_qtoll( const long double *a )`
: This function converts the extended precision value of `a` to a `signed long long` by truncating any fractional part and returns the `signed long long` value.

`unsigned long long _q_qtoull( const long double *a )`
: This function converts the extended precision value of `a` to an `unsigned long long` by truncating any fractional part and returns the unsigned `long long` value.

`long double _q_ulltoq( unsigned long long a )`
: This function converts the `unsigned long long` value of `a` to extended precision and returns the extended precision value.

`long double _q_neg( const long double *a )`

This function returns `-a` without raising any exceptions.

`double _q_qtod( const long double *a )`

This function converts the extended precision value of `a` to double precision and returns the double precision value.

`int _q_qtoi( const long double *a )`

This function converts the extended precision value of `a` to a signed integer by truncating any fractional part and returns the signed integer value.

`float _q_qtos( const long double *a )`

This function converts the extended precision value of `a` to single precision and returns the single precision value.

`unsigned int _q_qtou( const long double *a )`

This function converts the extended precision value of `a` to an unsigned integer by truncating any fractional part and returns the unsigned integer value.

`long double _q_sqrt( const long double *a )`

This function returns the square root of `a` computed to quadruple precision.

`long double _q_stoq( float a )`

This function converts the single precision value of `a` to extended precision and returns the extended precision value.

`long double _q_sub( const long double *a, const long double *b )`

This function returns `a - b` computed to extended precision.

`long double _q_utoq( unsigned int a )`

This function converts the unsigned integer value of `a` to extended precision and returns the extended precision value.

`unsigned int __dtou( double a )`

This function converts the double precision value of `a` to an unsigned integer by truncating any fractional part and returns the unsigned integer value. `__dtou` raises exceptions as follows:

- If `0 <= a < 2^32`, the operation is successful.
- If `a` is a whole number, no exceptions are raised.
- If `a` is not a whole number, the inexact exception is raised.

Otherwise, the value returned by `__dtou` is unspecified, and the invalid operation exception is raised. If any exceptions occur, the appropriate exception bits in the `FPSCR` are updated and, if the corresponding exception enable bits are set and the FE0 and FE1 bits of the MSR register are not both zero, the system floating-point exception trap handler is invoked.

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

`int _q_cmpe( const long double *a, const long double *b )`

This function performs an ordered comparison of the extended precision values of a and b and returns an integer value that indicates their relative ordering according to the same convention as _q_cmp.

`long double _q_div( const long double *a, const long double *b )`

This function returns a / b computed to extended precision.

`long double _q_dtoq( double a )`

This function converts the double precision value of a to quadruple precision and returns the extended precision value.

`int _q_feq( const long double *a, const long double *b )`

This function performs an unordered comparison of the extended precision values of a and b and returns a nonzero value if they are equal, zero otherwise.

`int _q_fge( const long double *a, const long double *b )`

This function performs an ordered comparison of the extended precision values of a and b and returns a nonzero value if a is greater than or equal to b, zero otherwise.

`int _q_fgt( const long double *a, const long double *b )`

This function performs an ordered comparison of the extended precision values of a and b and returns a nonzero value if a is greater than b, zero otherwise.

`int _q_fle( const long double *a, const long double *b )`

This function performs an ordered comparison of the extended precision values of a and b and returns a nonzero value if a is less than or equal to b, zero otherwise.

`int _q_flt( const long double *a, const long double *b )`

This function performs an ordered comparison of the extended precision values of a and b and returns a nonzero value if a is less than b, zero otherwise.

`int _q_fne( const long double *a, const long double *b )`

This function performs an unordered comparison of the extended precision values of a and b and returns a nonzero value if they are unordered or not equal, zero otherwise.

`long double _q_itoq( int a )`

This function converts the integer value of a to extended precision and returns the extended precision value.

`long double _q_mul( const long double *a, const long double *b )`

This function returns a * b computed to extended precision.

6 LIBRARIES

System Library

Support Routines

In addition to operating system services, `libsys` contains the following processor-specific support routines.

_q_add	_q_cmp	_q_cmpe	_q_div
_q_dtoq	_q_feq	_q_fge	_q_fgt
_q_fle	_q_flt	_q_fne	_q_itoq
_q_mul	_q_neg	_q_qtod	_q_qtoi
_q_qtos	_q_qtou	_q_sqrt	_q_stoq
_q_sub	_q_utoq	_ _dtou	

Figure 6-1 `libsys` Support Routines

Routines listed below employ the standard calling sequence described in **Function Calling Sequence** in Chapter 3. Descriptions are written from the caller's point of view with respect to register usage and stack frame layout.

Note that the functions prefixed by _q_ below implement extended precision arithmetic operations. The following restrictions apply to each of these functions:

- When a function returns an extended precision result, that result is rounded in accordance with the setting of the rounding control (`RN`) field of the `FPSCR` register.
- If any floating-point exceptions occur, the appropriate exception bits in the `FPSCR` are updated; if the corresponding exception is enabled, the floating-point exception trap handler is invoked.

Note – The references in the following descriptions to `a` and `b`, where the corresponding arguments are pointers to `long double` quantities, refer to the values pointed to, not the pointers themselves.

`long double _q_add( const long double *a, const long double *b )`
This function returns `a + b` computed to extended precision.

`int _q_cmp( const long double *a, const long double *b )`
This function performs an unordered comparison of the extended precision values of `a` and `b` and returns an integer value that indicates their relative ordering, as shown below.

Following the steps below, the dynamic linker and the program cooperate to resolve symbolic references through the procedure linkage table. Again, the steps described below are for explanation only. The precise execution-time behavior of the dynamic linker is not specified.

1. As shown above, all procedure linkage table entries initially transfer to `.PLTresolve`, allowing the dynamic linker to gain control at the first execution of each table entry. For example, assume the program calls *name*, which transfers control to the label `.PLTi`. The procedure linkage table entry loads into `r11` four times the index of the relocation entry for `.PLTi` and branches to `.PLTresolve`, which then calls into the dynamic linker with a pointer to the symbol table for the object in `r12`.
2. The dynamic linker finds relocation entry `i` corresponding to the index in `r11`. It will have type `R_PPC_JMP_SLOT`, its offset will specify the address of `.PLTi`, and its symbol table index will reference *name*.
3. Knowing this, the dynamic linker finds the symbol's "real" value. It then modifies the code at `.PLTi` in one of two ways. If the target symbol is reachable from `.PLTi` by a branch instruction, it overwrites the `"addi r11,r0,4*(i-1)"` instruction at `.PLTi` with a branch to the target. On the other hand, if the target symbol is not reachable from `.PLTi`, the dynamic linker loads the target address into word `.PLTtable+4*(i-1)` and overwrites the `"b .PLTresolve"` with a `"b .PLTcall"`.
4. Subsequent executions of the procedure linkage table entry will transfer control directly to the function, either directly or by using `.PLTcall`, without invoking the dynamic linker.

For PLT indexes greater than or equal to `2^13`, only the even indexes shall be used and four words shall be allocated for each entry. If the above scheme is used, this allows four instructions for loading the index and branching to `.PLTresolve` or `.PLTcall`, instead of only two.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker resolves the function call binding at load time, before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_PPC_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: 1) The initial reference to a shared object function takes longer than subsequent calls because the dynamic linker intercepts the call to resolve the symbol, and some applications cannot tolerate this unpredictability. 2) 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.

---

on. The relocation type for each entry shall be `R_PPC_JMP_SLOT`, the relocation offset shall specify the address of the first byte of the associated procedure linkage table entry, and the symbol table index shall reference the appropriate symbol.

To illustrate procedure linkage tables, Figure 5-3 shows how the dynamic linker might initialize the procedure linkage table when loading the executable or shared object.

```
.PLT:

.PLTresolve:
    addis   r12,r0,dynamic_linker@ha
    addi    r12,r12,dynamic_linker@l
    mtctr   r12
    addis   r12,r0,symtab_addr@ha
    addi    r12,r12,symtab_addr@l
    bctr
.PLTcall:
    addis   r11,r11,.PLTtable@ha
    lwz     r11,.PLTtable@l(r11)
    mtctr   r11
    bctr
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
.PLT1:
    addi    r11,r0,4*0
    b       .PLTresolve
                  . . .
.PLTi:
    addi    r11,r0,4*(i-1)
    b       .PLTresolve
                  . . .
.PLTN:
    addi    r11,r0,4*(N-1)
    b       .PLTresolve

.PLTtable:
    <N word table begins here>
```

Figure 5-3 Procedure Linkage Table Example

for details.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 these rules:

- 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.
- 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.
- 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. The dynamic linker determines the destinations' absolute addresses and modifies the procedure linkage table's memory image accordingly. The dynamic linker can thus 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.

For the PowerPC, the procedure linkage table (the `.plt` section) is not initialized in the executable or shared object file. Instead, the link editor simply reserves space for it, and the dynamic linker initializes it and manages it according to its own, possibly implementation-dependent needs, subject to the following constraints:

- The first 18 words (72 bytes) of the procedure linkage table are reserved for use by the dynamic linker. There shall be no branches from the executable or shared object into these first 18 words.
- If the executable or shared object requires `N` procedure linkage table entries, the link editor shall reserve `3*N` words (`12*N` bytes) following the 18 reserved words. The first `2*N` of these words are the procedure linkage table entries themselves. The static linker directs calls to bytes (`72 + (i-1)*8`), for `i` between `1` and `N` inclusive. The remaining `N` words (`4*N` bytes) are reserved for use by the dynamic linker.

As mentioned before, a relocation table is associated with the procedure linkage table. The `DT_JMPREL` entry in the `_DYNAMIC` array gives the location of the first relocation entry. The relocation table's entries parallel the procedure linkage table entries in a one-to-one correspondence. That is, relocation table entry 1 applies to procedure linkage table entry 1, and so

When the dynamic linker creates memory segments for a loadable object file, it processes the relocation entries, some of which will be of type `R_PPC_GLOB_DAT`, referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the global offset 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.

A global offset table entry provides direct access to the absolute address of a symbol without compromising position-independence and sharability. Because the executable file and shared objects have separate global offset tables, a symbol 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 dynamic linker 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 PowerPC, the symbol `_GLOBAL_OFFSET_TABLE_` may be used to access the table. The symbol may reside in the middle of the `.got` section, allowing both positive and negative "subscripts" into the array of addresses. Four words in the global offset table are reserved:

- The word at `_GLOBAL_OFFSET_TABLE_[-1]` shall contain a `blrl` instruction (see the text relating to Figure 3-33, "Prologue and Epilogue Sample Code").
- The word at `_GLOBAL_OFFSET_TABLE_[0]` is set by the link editor 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.
- The word at `_GLOBAL_OFFSET_TABLE_[1]` is reserved for future use.
- The word at `_GLOBAL_OFFSET_TABLE_[2]` is reserved for future use.

The global offset table resides in the ELF `.got` section.

Function Addresses

References to the address of a function from an executable file and the shared objects associated with it need to 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

Table 5-2 Shared Object Segment Example

Source	Text	Data	Base Address
File	`0x000200`	`0x02a400`	
Process 1	`0x100200`	`0x12a400`	`0x100000`
Process 2	`0x200200`	`0x22a400`	`0x200000`
Process 3	`0x300200`	`0x32a400`	`0x300000`
Process 4	`0x400200`	`0x42a400`	`0x400000`

Program Interpreter

A program shall not specify a program interpreter other than `/usr/lib/ld.so.1`.

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` This entry's `d_ptr` member gives the address of the first byte in the procedure linkage table (`.PLT` in Figure 5-3).

`DT_JMPREL` As explained in the *System V ABI*, this entry is associated with a table of relocation entries for the procedure linkage table. For the PowerPC, this entry is mandatory both for executable and shared object files. Moreover, the relocation table's entries must have a one-to-one correspondence with the procedure linkage table. The table of `DT_JMPREL` relocation entries is wholly contained within the `DT_RELA` referenced table. See **Procedure Linkage Table** later in this chapter for more 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.

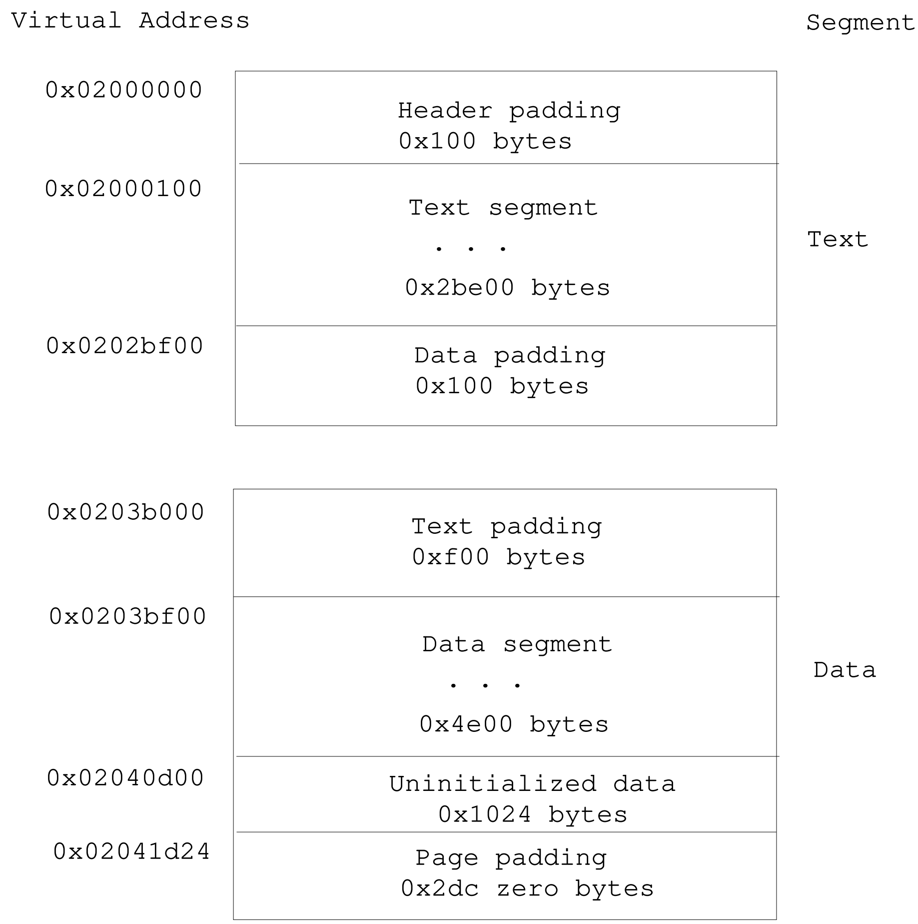

Figure 5-2 Process Image Segments

One aspect of segment loading differs between executable files and shared objects. Executable file segments may contain absolute code. For the process to execute correctly, the segments must reside at the virtual addresses assigned when building the executable file, with the system using the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This allows a segment's virtual address to change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the "relative positions" of the segments. 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. Table 5-2 shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

Table 5-1 Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x02000100	0x0203bf00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W
p_align	0x10000	0x10000

Although the file offsets and virtual addresses are congruent modulo 64 Kbytes for both text and data, up to four file pages can 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 may hold a copy of the beginning of data.
- The first data page may have 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 memory permissions as if each segment were complete and separate; segment addresses are adjusted to ensure that each logical page in the address space has a single set of permissions. In the example in Figure 5-1, the file region holding the end of text and the beginning of data is 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 the last data page of a file includes information not in the logical memory page, the extraneous data must be set to zero, rather than to 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 the program in Figure 5-1 is presented in Figure 5-2, assuming 4096 (`0x1000`) byte pages.

5 PROGRAM LOADING AND DYNAMIC LINKING

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 offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for the PowerPC processor family segments are congruent modulo 64 Kbytes (`0x10000`) or larger powers of 2. Although 4096 bytes is currently the PowerPC page size, this allows files to be suitable for paging even if implementations appear with larger page sizes. The value of the `p_align` member of each program header in a shared object file must be `0x10000`. Figure 5-1 is an example of an executable file assuming an executable program linked with a base address of `0x02000000` (32 Mbytes).

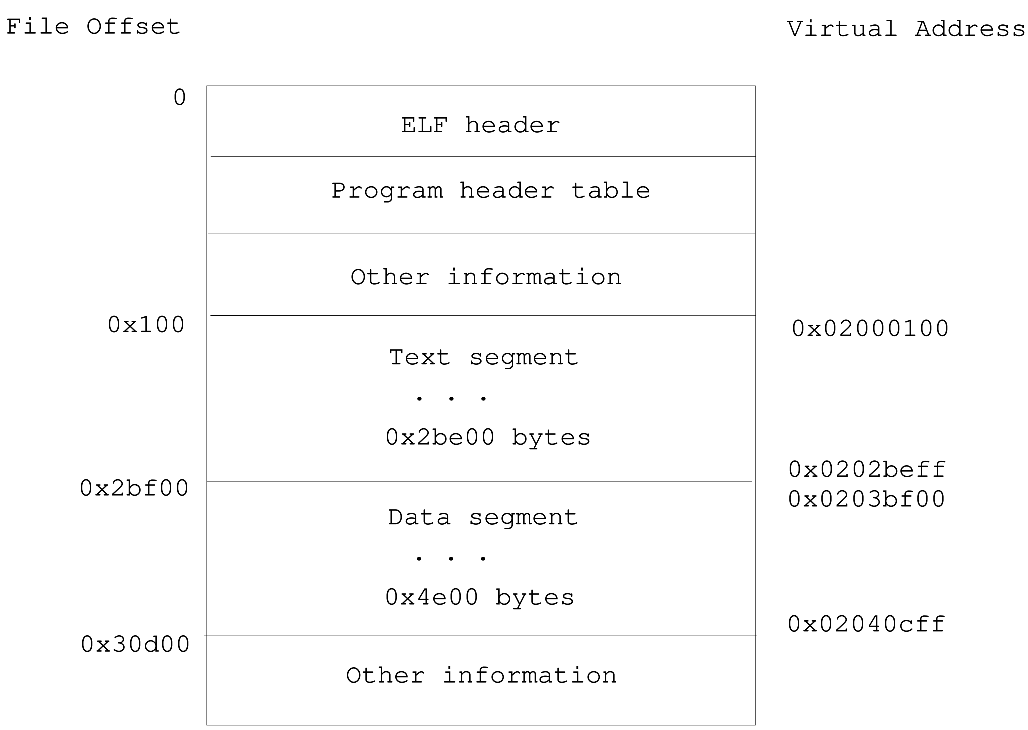

Figure 5-1 Executable File Example

Name	Description
R_PPC_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_PPC_GLOB_DAT	This relocation type resembles R_PPC_ADDR32, except that it sets a global offset table entry to the address of the specified symbol. This special relocation type allows one to determine the correspondence between symbols and global offset table entries.
R_PPC_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 Figure 5-3 in Chapter 5).
R_PPC_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_PPC_LOCAL24PC	This relocation type resembles R_PPC_REL24, except that it uses the value of the symbol within the object, not an interposed value, for S 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.
R_PPC_UADDR*	These relocation types are the same as the corresponding R_PPC_ADDR* types, except that the datum to be relocated is allowed to be unaligned.

Table 4-8 Relocation Types *(Continued)*

Name	Value	Field	Calculation
R_PPC_PLTREL32	28	word32	L + A − P
R_PPC_PLT16_LO	29	half16	#lo(L + A)
R_PPL_PLT16_HI	30	half16	#hi(L + A)
R_PPC_PLT16_HA	31	half16	#ha(L + A)
R_PPC_SDAREL16	32	half16*	S + A − _SDA_BASE_
R_PPC_SECTOFF	33	half16*	R + A
R_PPC_SECTOFF_LO	34	half16	#lo(R + A)
R_PPC_SECTOFF_HI	35	half16	#hi(R + A)
R_PPC_SECTOFF_HA	36	half16	#ha(R + A)
R_PPC_ADDR30	37	word30	(S + A − P) >> 2

Relocation values not in Table 4-8 and less than 101 or greater than 200 are reserved. Values in the range 101-200 and names beginning with "R_PPC_EMB_" have been assigned for embedded system use.

The relocation types whose Field column entry contains an asterisk * are subject to failure if the value computed does not fit in the allocated bits.

The relocation types in which the names include _BRTAKEN or _BRNTAKEN specify whether the branch prediction bit (bit 10) should indicate that the branch will be taken or not taken, respectively. For an unconditional branch, the branch prediction bit must be 0.

Relocation types with special semantics are described below.

Name	Description
R_PPC_GOT16*	These relocation types resemble the corresponding R_PPC_ADDR16* types, except that they refer to the address of the symbol's global offset table entry and additionally instruct the link editor to build a global offset table.
R_PPC_PLTREL24	This relocation type refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table. There is an implicit assumption that the procedure linkage table for a module will be within +/- 32 Mbytes of an instruction that branches to it, so that the R_PPC_PLTREL24 relocation type is the only one needed for relocating branches to procedure linkage table entries.

Table 4-8 Relocation Types

Name	**Value**	**Field**	**Calculation**
R_PPC_NONE	0	none	none
R_PPC_ADDR32	1	word32	S + A
R_PPC_ADDR24	2	low24*	(S + A) >> 2
R_PPC_ADDR16	3	half16*	S + A
R_PPC_ADDR16_LO	4	half16	#lo(S + A)
R_PPC_ADDR16_HI	5	half16	#hi(S + A)
R_PPC_ADDR16_HA	6	half16	#ha(S + A)
R_PPC_ADDR14	7	low14*	(S + A) >> 2
R_PPC_ADDR14_BRTAKEN	8	low14*	(S + A) >> 2
R_PPC_ADDR14_BRNTAKEN	9	low14*	(S + A) >> 2
R_PPC_REL24	10	low24*	(S + A − P) >> 2
R_PPC_REL14	11	low14*	(S + A − P) >> 2
R_PPC_REL14_BRTAKEN	12	low14*	(S + A − P) >> 2
R_PPC_REL14_BRNTAKEN	13	low14*	(S + A − P) >> 2
R_PPC_GOT16	14	half16*	G + A
R_PPC_GOT16_LO	15	half16	#lo(G + A)
R_PPC_GOT16_HI	16	half16	#hi(G + A)
R_PPC_GOT16_HA	17	half16	#ha(G + A)
R_PPC_PLTREL24	18	low24*	(L + A − P) >> 2
R_PPC_COPY	19	none	none
R_PPC_GLOB_DAT	20	word32	S + A
R_PPC_JMP_SLOT	21	none	see below
R_PPC_RELATIVE	22	word32	B + A
R_PPC_LOCAL24PC	23	low24*	see below
R_PPC_UADDR32	24	word32	S + A
R_PPC_UADDR16	25	half16*	S + A
R_PPC_REL32	26	word32	S + A − P
R_PPC_PLT32	27	word32	L + A

P — Represents the place (section offset or address) of the storage unit being relocated (computed using r_offset).

R — Represents the offset of the symbol within the section in which the symbol is defined (its section-relative address).

S — Represents the value of the symbol whose index resides in the relocation entry.

Relocation entries apply to halfwords or words. In either case, the 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 PowerPC family uses only the Elf32_Rela relocation entries with explicit addends. For the relocation entries, the r_addend member serves as the relocation addend. In all cases, the offset, addend, and the computed result use the byte order specified in the ELF header.

The following general rules apply to the interpretation of the relocation types in Table 4-8:

- "+" and "-" denote 32-bit modulus addition and subtraction, respectively. ">>" denotes arithmetic right-shifting (shifting with sign copying) of the value of the left operand by the number of bits given by the right operand.
- For relocation types in which the names contain "14" or "16," the upper 17 bits of the value computed before shifting must all be the same. For relocation types whose names contain "24," the upper 7 bits of the value computed before shifting must all be the same. For relocation types whose names contain "14" or "24," the low 2 bits of the value computed before shifting must all be zero.
- #hi(*value*) and #lo(*value*) denote the most and least significant 16 bits, respectively, of the indicated value. That is, #lo(*x*) = (*x* & 0xFFFF) and #hi(*x*) = ((*x* >> 16) & 0xFFFF). The "high adjusted" value, #ha (*value*), compensates for #lo() being treated as a signed number: #ha(*x*) = (((*x* >> 16) + ((*x* & 0x8000) ? 1 : 0)) & 0xFFFF).
- Reference in a calculation to the value G implicitly creates a GOT entry for the indicated symbol.
- _SDA_BASE_ is a symbol defined by the link editor whose value in shared objects is the same as _GLOBAL_OFFSET_TABLE_, and in executable programs is an address within the small data area. See **Small Data Area** above.

`word32` This specifies a 32-bit field occupying 4 bytes, the alignment of which is 4 bytes unless otherwise specified.

`word30` This specifies a 30-bit field contained within bits 0-29 of a word with 4-byte alignment. The two least significant bits of the word are unchanged.

`low24` This specifies a 24-bit field contained within a word with 4-byte alignment. The six most significant and the two least significant bits of the word are ignored and unchanged (for example, "Branch" instruction).

`low14` This specifies a 14-bit field contained within a word with 4-byte alignment, comprising a conditional branch instruction. The 14-bit relative displacement in bits 16-29, and possibly the "branch prediction bit" (bit 10), are altered; all other bits remain unchanged.

`half16` This specifies a 16-bit field occupying 2 bytes with 2-byte alignment (for example, the immediate field of an "Add Immediate" instruction).

Calculations in Table 4-8 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 determines how to combine and locate the input files, next it updates the symbol values, and then it performs relocations.

Relocations applied to executable or shared object files are similar and accomplish the same result. The following notations are used in Table 4-8:

`A` Represents the addend used to compute the value of the relocatable field.

`B` Represents 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` Represents 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.

`L` Represents the 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.

Relocation

Relocation Types

Relocation entries describe how to alter the instruction and data relocation fields shown in Figure 4-1 (bit numbers appear in the lower box corners; Little-Endian byte numbers appear in the upper right box corners; Big-Endian numbers appear in the upper left box corners).

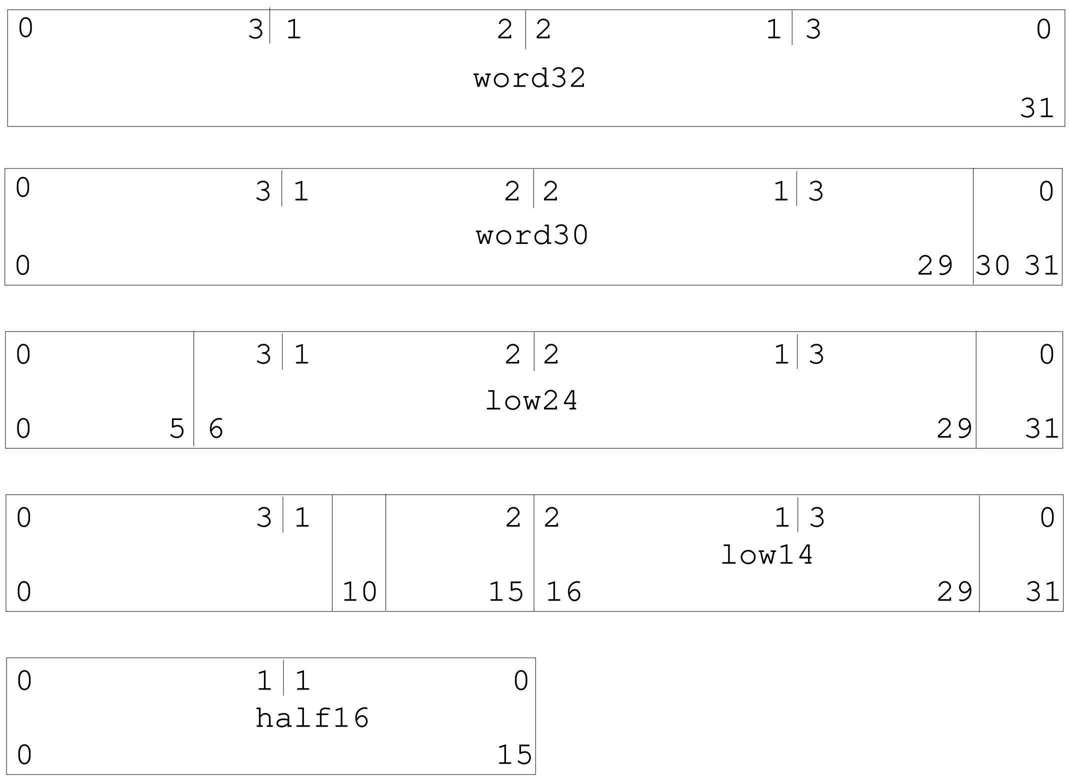

Figure 4-1 Relocation Fields

Note, however, that the size of the small data area is limited, as indicated above. Compilers that support small data area relative addressing determine whether or not an eligible data item is placed in the small data area based on its size. All data items less than or equal to a specified size (the default is usually 8 bytes) are placed in the small data area. Initialized data items are placed in a `.sdata` section, uninitialized data items in a `.sbss` section. If the default size results in a small data area that is too large to be addressed with 16-bit relative offsets, the link editor fails to build the executable or shared object, and some of the code that makes up the file must be recompiled with a smaller value for the size criterion.

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 the file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This informs 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 nonzero, the value is 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.

Small Data Area

The small data area is part of the data segment of an executable program. It contains data items within the `.sdata` and `.sbss` sections, which can be addressed with 16-bit signed offsets from the base of the small data area.

In both shared object and executable files, the small data area straddles the boundary between initialized and uninitialized data in the data segment of the file. The usual order of sections in the data segment, some of which may be empty, is:

```
.data
.got
.sdata
.sbss
.plt
.bss
```

Only data items with local (non-global) scope may appear in the small data area of a shared object. In a shared object the small data area follows the global offset table, so data in the small data area can be addressed relative to the GOT pointer. However, in this case, the small data area is limited in size to no more than 32 Kbytes, and less if the global offset table is large.

For executable files, up to 64 Kbytes of data items with local or global scope can be placed into the small data area. In an executable file, the symbol `_SDA_BASE_` (small data area base) is defined by the link editor to be an address relative to which all data in the `.sdata` and `.sbss` sections can be addressed with 16-bit signed offsets or, if there is neither a `.sdata` nor a `.sbss` section, the value 0. In a shared object, `_SDA_BASE_` is defined to have the same value as `_GLOBAL_OFFSET_TABLE_`. The value of `_SDA_BASE_` in an executable is normally loaded into `r13` at process initialization time, and `r13` thereafter remains unchanged. In particular, shared objects shall not change the value in `r13`.

Compilers may generate "short-form," one-instruction references for all data items that are in the `.sdata` or `.sbss` sections. In executable files, such references are relative to `r13`; in shared objects, they are relative to a register that contains the address of the global offset table. Placing more data items in small data areas usually results in smaller and faster program execution.

Similarly, a module (conventionally, `crtn.o`), whose contents reflect the pseudo-code in Figure 4-3, is appended after any object modules containing executable instructions or tags.

```
            .section        .tags
_tag_end:

            .section        .module_tags
            .long  0             # lastpc
            .long  _tag_end      # lasttag
_module_tags_end:

            .section        .fini
            ...
            # Call __delete_module_tags(_module_tags_end-4*6)
            ...
```

Figure 4-3 `crtn.o` Pseudo-code

When the link editor builds the executable or shared object, it concatenates the contributions to each section from the various objects, in order. Therefore, assuming that only `crti.o` and `crtn.o` contribute to the `.module_tags` section, the link editor places the two words in the `.module_tags` section of the `crtn.o` module immediately after the four words in that section in the `crti.o` module, forming a complete, six-word `module_tags` structure. The code in the `.init` section, which makes up part of the initialization function specified in the dynamic section, adds the tag section described by the `module_tags` to the list of active tag sections. The code in the `.fini` section deletes the `module_tags` from the active list using addressing relative to the end of the structure to avoid the need for a globally visible, but not unique, symbol.

The mechanism for locating tags involves three functions in the C library (See **Required Routines** in Chapter 6):

`__add_module_tags(struct module_tags *mt)`
: Adds the `module_tags` for an object to the list of module tags for the process.

`__delete_module_tags(struct module_tags *mt)`
: Removes the `module_tags` for an object from the list.

`__tag_lookup_pc(caddr_t pc)`
: Returns a pointer to the `module_tags` structure that describes the tags section applicable to the given PC value, or NULL if there is no applicable tags section.

The `__add_module_tags` function is generally called from the initialization function specified in the dynamic structure; the `__delete_module_tags` function is generally called from the termination function. (See **Initialization and Termination Functions** in the *System V Application Binary Interface*.)

The remainder of this section describes one way to arrange for a module to construct its `module_tags` structure and add it to the list of active tag sections.

In constructing an executable or shared object, a module (conventionally, `crti.o`), whose contents reflect the pseudo-code in Figure 4-2 below, is inserted before any object modules containing executable instructions or tags.

```
            .section        .tags
_tag_start:

            .section        .module_tags
_module_tags_start:
            .long   0            # next pointer
            .long   0            # prev pointer
            .long   0            # firstpc
            .long   _tag_start   # firsttag

            .section.init
            ...
            # Call __add_module_tags(_module_tags_start)
            ...
```

Figure 4-2 `crti.o` Pseudo-code

7. [REGISTERS VALID TAG] Obtain FR and GR, which define the register save areas, from the closest Frame or Frame Valid tag with a BASE less than or equal to that of the Registers Valid tag. Replace the recorded values of the floating-point (resp., general) registers corresponding to 1's in FLOAT_REGS (resp., GEN_REGS) with the values in the register save areas in the frame addressed by SP. Replace the recorded values of LR and CR according to C_REG and LR_INREG as in step 5, then continue with step 8 [SNAPSHOT].

8. [SNAPSHOT] The recorded values are those at entry to the current function, which was called from the address in PC. The caller's frame is pointed to by the value recorded in SP. Replace SP with this value and continue with step 9.

9. [POP FRAME] If the recorded SP is nonzero, continue with step 4 [LOOP]. Otherwise, this is the end of the call chain. Terminate.

Locating Tags

Each object (executable file or shared object) in a process image contains the tags that apply to its executable instructions. The preceding section assumed that it was possible for a program to find the tag, if any, associated with a particular address. This section describes the mechanism that a program uses to locate its tags.

The tags for a process are described by a doubly linked list of module_tags structures as shown in Figure 4-1 below. There is usually one such structure for each module in the process.

```
struct module_tags {
  struct module_tags *next;    /* Next entry in list*/
  struct module_tags *prev;    /* Previous entry in list */
  caddr_t firstpc;             /* First PC to which applicable */
  caddr_t firsttag;            /* Beginning of tags */
  caddr_t lastpc;              /* Last PC to which applicable */
  caddr_t lasttag;             /* First address beyond end of tags */
};
```

Figure 4-1 module_tags Structure

The firstpc and lastpc values may be zero when __add_module_tags is called. In this case, __tag_lookup_pc can compute the values by finding the PC range implied by tags addressed by firsttag and lasttag.

Table 4-7 Special Tag Format

Word	Bits	Name	Description
1	0-29	BASE_OFFSET	The (signed) number of words between the tag and the BASE to which it refers, positive if the tag is at a lower address than BASE.
1	30-31	TYPE	3
2	0-3	LR_SAVEREG	The (volatile) register that contains the value of the link register at function entry.
2	4-19	RESERVED	0
2	20-29	RANGE	The (unsigned) number of words between the first and last word to which the tag applies (zero if the tag applies to only one word).
2	30-31	RESERVED	0

Stack Traceback Using Tags

The following algorithm reconstructs the values in the nonvolatile registers at the entry to all functions in the call chain. It assumes an image of the stack, the values in the registers, and the address of the next instruction to be executed (PC). The algorithm creates a snapshot of the register values at each function entry, beginning with the latest and working backward through successive call sites.

1. [INITIALIZE] Record the values in the nonvolatile general and floating-point registers, CR, LR, and SP.
2. [NO TAG] If there is no tag associated with PC, then the recorded values are those at entry to the function. Set PC to the value recorded for the LR. Go to step 8 [SNAPSHOT].
3. [SPECIAL TAG] If the tag associated with PC is a Special tag, then the recorded values are those at entry to the function, but the address of the caller, the LR on entry, is in the LR_SAVEREG specified in the tag. Set PC to this value and go to step 8 [SNAPSHOT}.
4. [LOOP] If the tag associated with PC is a:
   - Frame Valid tag, continue with step 5.
   - Frame Tag, continue with step 6.
   - Registers Valid tag, continue with step 7.
5. [FRAME VALID TAG] If PC lies beyond the BASE for the tag, then replace the recorded values of the FV (resp., GV) highest numbered floating-point (resp., general) registers with the values in the register save areas in the frame addressed by SP, and if C_REG is 1, replace its recorded value. If LR_INREG is 0, replace the recorded value of LR with the saved LR value in the frame pointed to by SP. Go to step 8 [SNAPSHOT].
6. [FRAME TAG] Same as for step 5, but with FV replaced with FR and GV replaced with GR.

Table 4-6 Registers Valid Tag Format

Word	Bits	Name	Description
1	0-17	FLOAT_REGS	One bit for each nonvolatile floating-point register, bit 0 for f31, ..., bit 17 for f14, with a 1 signifying that the register is saved in the register save area.
1	18-29	START_OFFSET	The number of words between the BASE of the nearest preceding Frame or Frame Valid tag and the first instruction to which this tag applies.
1	30-31	TYPE	2
2	0-17	GEN_REGS	One bit for each nonvolatile general register, bit 0 for r31, ..., bit 17 for r14, with a 1 signifying that the register is saved in the register save area.
2	18-29	RANGE	The number of words between the first and the last instruction to which this tag applies.
2	30	C_REG	1 if and only if the condition register is saved in its assigned place in the register save area.
2	31	RESERVED	0

The only Special tag defined in this version of tags applies to leaf functions which, though they do not need to establish a frame, must modify the value in the link register. For example, a leaf function in a shared object that needs no frame but requires a pointer to the global offset table may use the following sequence of instructions to access static data via the global offset table.

```
func:     mflr      r11                        #Save LR in r11
          bl        _GLOBAL_OFFSET_TABLE_-4    #GOT pointer to link register

tbase:    mflr      r12                        #GOT pointer to r12
          ...                                  #Use r12
          mtlr      r11                        #Restore LR
tend:     blr                                  #Return
```

The above code would have a Special tag specifying an LR_SAVEREG of 11 with a BASE referencing the word at tbase (the first instruction for which the LR does not contain its value on entry) and a RANGE of ((tend - tbase)/4) - 1.

Table 4-4 Frame Tag Format *(Continued)*

Word	Bits	Name	Description
2	16-29	RANGE	The (unsigned) number of words between BASE+4*FRAME_START and the last word to which the tag applies. A tag ceases to apply at the instruction after the one that deallocates the frame, and earlier if the register save state changes such that it requires another tag. A RANGE of 0 implies only the establishing of a frame and the sizes of the save areas; subsequent Registers Valid tags supply the register save data.
2	30	C_REG	1 if and only if the condition register is saved in its assigned place in the register save area.
2	31	LR_INREG	1 if the link register holds its contents on entry to the function and is not saved in the LR save word of the previous frame.

Table 4-5 Frame Valid Tag Format

Word	Bits	Name	Description
1	0-5	FRAME_START	Same as Frame tag, except that FV and GV specify the number of registers saved.
1	6-29	BASE_OFFSET	Same as Frame tag.
1	30-31	TYPE	1
2	0-4	FV	The number of nonvolatile floating-point registers saved in the floating-point registers save area within the RANGE of the tag. If a given floating-point register is saved, so must all others with higher numbers.
2	5-9	FR	Same as Frame tag.
2	10-14	GV	The number of nonvolatile general registers saved in the general registers save area within the RANGE of the tag. If a given register is saved, so must all others with higher numbers.
2	15-19	GR	Same as Frame tag.
2	20-29	RANGE	Same as Frame tag.
2	30	C_REG	Same as Frame tag.
2	31	LR_INREG	Same as Frame tag.

Tag Formats

Every tag consists of two words (8 bytes). The low-order 2 bits of the first word of each tag specify the tag type, encoded as shown in Table 4-3.

Table 4-3 Tag Formats

Tag Code	Tag Type
`0`	Frame
`1`	Frame Valid
`2`	Registers Valid
`3`	Special

Tables 4-5, 4-5, 4-6, and 4-7 specify the formats of each tag type. For the Frame, Frame Valid, and Special tags, `BASE` refers to the address within the code relative to which offset fields within the tag are computed and on which the tags are sorted. `BASE` usually refers to the first instruction following the instruction that establishes the frame. For Frame tags, a `RANGE` of 0 implies only the establishing of a frame and the sizes of the save areas; subsequent Registers Valid tags supply the register save data.

Table 4-4 Frame Tag Format

Word	Bits	Name	Description
1	0-29	`BASE_OFFSET`	The (signed) number of words between the tag and the `BASE` to which it refers, positive if the tag is at a lower address than the `BASE`.
1	30-31	`TYPE`	0
2	0-5	`FRAME_START`	The (unsigned, possibly zero) number of words between `BASE` and the first address at which registers implied by the values `FR` and `GR` have been saved. In the interval between that address and `BASE`, a frame has been established, and the `LR` save word of the previous frame contains the address from which the function was called, but the nonvolatile registers still contain their values when the function was entered.
2	6-10	`FR`	Size in double words of the floating-point register save area.
2	11-15	`GR`	Size in words of the general register save area.

Tags

Tag Overview

Tags facilitate determining the contents of nonvolatile registers as they were when a function was entered. Given the address of the next instruction to be executed, and the tag, if any, applicable to that address, a debugger or exception handler can determine the register contents upon function entry.

The stack frame layout, and in particular the register save areas within a frame, are specified in Chapter 3. Tags make it possible to determine which stack frame is associated with a section of code and which nonvolatile registers at the time of entry to the function are within the register save areas rather than in the registers themselves.

In the simplest case, a leaf function needs no tag if it 1) does not establish its own frame and 2) does not disturb the contents of any of the nonvolatile registers or the link register. Similarly, within a function, code that is leaf-like (in that it has not yet established a frame or has restored the stack, nonvolatile registers, and the link register to their state on entry to the function) needs no tag.

There are four tag formats as defined in Table 4-3 and described in subsequent tables.

A function that establishes a frame requires at least one "Frame" or "Frame Valid" tag. Both of these formats specify the point in the code at which a frame is established and the sizes of the general and floating-point register save areas. They may also specify a point at which a contiguous set of general and floating-point registers have been saved in the save area and a range of addresses from that point within which the frame and the saved registers remain valid. A function requires only a Frame or Frame Valid tag if it 1) establishes a frame, 2) saves all the nonvolatile registers that it uses before changing any of them, and then 3) restores the registers and deallocates the frame. The differences between the Frame and Frame Valid tags are:

- A Frame tag can support much larger modules. A Frame tag can be up to 2 Gbytes away from the text to which it refers, while a Frame Valid tag must be within 32 Mbytes of the text.
- A Frame tag can cover a range of up to 16,384 instructions, while a Frame Valid tag can cover only 1024 instructions. Functions with frames that span more instructions may require multiple Frame or Frame Valid tags.
- A Frame tag requires that all the registers for which space has been allocated in the save areas be saved. A Frame Valid tag can specify not only the save area sizes but a subset of the registers that are stored in the save area within the region covered by the tag.

Functions that intersperse saving some nonvolatile registers with using other nonvolatile registers, or which save and use higher-numbered nonvolatile general or floating-point registers before saving lower-numbered registers, need to use "Registers Valid" tags in addition to one or more Frame or Frame Valid tags. A Registers Valid tag specifies a range of addresses for which the tag is valid and 1 bit for each nonvolatile general and floating-point register indicating whether it has been saved in the register save area and may not contain its value on entry.

Finally, there is a "Special" tag for functions that establish no frame but use the link register.

Special sections are described below.

Name	Description
`.got`	This section holds the Global Offset Table, or GOT. 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.
`.sdata`	This section holds initialized small data that contribute to the program memory image. See **Small Data Area** later in this chapter for details.
`.sbss`	This section holds uninitialized small data that contribute to the program memory image. The system sets the data to zeros when the program begins to run. See **Small Data Area** for details.
`.tags`	This section contains tags as described in **Tags** below. The size (`sh_entsize`) of each entry in this section is 8 and the alignment (`sh_addralign`) is 4. The relocation section `.rela.tags`, associated with the `.tags` section, should have the `SHF_EXCLUDE` attribute.
`.taglist`	This section contains data that enable a program to locate its tags. Locating tags is described in **Tags** below.
`.tagsym`	This section, which appears in object files only (not executable or shared objects), contains one entry for each entry in the `.tags` section. Each entry has `STB_LOCAL` binding and is of type `STT_NOTYPE`. The `st_shndx` and `st_value` fields of the entries specify the index of the section and the section offset to which the tag applies, respectively.

Note – The PowerPC Embedded ABI shares most of the linkage conventions and ELF file structuring conventions of this ABI. However, section names beginning with the string "`.PPC.EMB.`", the section names `.sdata2` and `.sbss2`, and the symbol `_SDA2_BASE` are reserved for the Embedded ABI.

Sections

Special Sections

Various sections hold program and control information. The sections listed in Table 4-2 are used by the system and have the types and attributes shown.

Note – The `.plt` section on the PowerPC is of type `SHT_NOBITS`, not `SHT_PROGBITS` as on most other processors.

Note – The `SHT_ORDERED` section type specifies that the link editor is to sort the entries in this section based on the sum of the symbol and addend values specified by the associated relocation entries. Entries without associated relocation entries shall be appended to the end of the section in an unspecified order. `SHT_ORDERED` is defined as `SHT_HIPROC`, the first value reserved in the System V ABI for processor-specific semantics.

Note –The `SHF_EXCLUDE` flag specifies that the link editor is to exclude this section from executable and shared objects that it builds when those objects are not to be further relocated. `SHF_EXCLUDE` has the value `0x80000000`.

Table 4-2 Special Sections

Name	**Type**	**Attributes**
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.plt	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_EXECINSTR
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.sbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.tags	SHT_ORDERED	SHF_ALLOC
.taglist	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.tagsym	SHT_SYMTAB	SHF_EXCLUDE

4 OBJECT FILES

ELF Header

Machine Information

For file identification in `e_ident`, the PowerPC processor family requires the values shown in Table 4-1.

Table 4-1 PowerPC Identification, `e_ident []`

Position	Value	Comments
`e_ident[EI_CLASS]`	ELFCLASS32	For all 32-bit implementations
`e_ident[EI_DATA]`	ELFDATA2MSB	For all Big-Endian implementations
`e_ident[EI_DATA]`	ELFDATA2LSB	For all Little-Endian implementations

The ELF header's `e_flags` member holds bit flags associated with the file. Since the PowerPC processor family defines no flags, this member contains zero.

The name `EF_PPC_EMB` and the value `0x80000000` are reserved for use in embedded systems.

Processor identification resides in the ELF header's `e_machine` member and must have the value 20, defined as the name `EM_PPC`.

Table 3-8 PowerPC Privileged Register Number Mapping *(Continued)*

Register Name	Number	Abbreviation
Data BAT Register 3 Upper	642	DBAT3U or SPR542
Data BAT Register 3 Lower	643	DBAT3L or SPR543
* Hardware Implementation Register 0	1108	HID0 or SPR1008
* Hardware Implementation Register 1	1109	HID1 or SPR1009
* Hardware Implementation Register 2	1110	HID2 or IABR or SPR1010
* Hardware Implementation Register 5	1113	HID5 or DABR or SPR1013
* Hardware Implementation Register 15	1123	HID15 or PIR or SPR1023

Address Class Codes

The PowerPC processor family defines the address class codes described in Table 3-9.

Table 3-9 Address Class Code

Code	Value	Meaning
ADDR_none	0	No class specified

Table 3-8 PowerPC Privileged Register Number Mapping

Register Name	Number	Abbreviation
Machine State Register	66	MSR
Segment Register 0-15	70–85	SR0–SR15
Data Storage Interrupt Status Register	118	DSISR or SPR18
Data Address Register	119	DAR or SPR19
Decrementer	122	DEC or SPR22
Storage Description Register 1	125	SDR1 or SPR25
Machine Status Save/Restore Register 0	126	SRR0 or SPR26
Machine Status Save/Restore Register 1	127	SRR1 or SPR27
Software-use Special Purpose Register 0	372	SPRG0 or SPR272
Software-use Special Purpose Register 1	373	SPRG1 or SPR273
Software-use Special Purpose Register 2	374	SPRG2 or SPR274
Software-use Special Purpose Register 3	375	SPRG3 or SPR275
Address Space Register	380	ASR or SPR280
External Access Register	382	EAR or SPR282
Time Base	384	TB or SPR284
Time Base Upper	385	TBU or SPR285
Processor Version Register	387	PVR or SPR287
Instruction BAT Register 0 Upper	628	IBAT0U or SPR528
Instruction BAT Register 0 Lower	629	IBAT0L or SPR529
Instruction BAT Register 1 Upper	630	IBAT1U or SPR530
Instruction BAT Register 1 Lower	631	IBAT1L or SPR531
Instruction BAT Register 2 Upper	632	IBAT2U or SPR532
Instruction BAT Register 2 Lower	633	IBAT2L or SPR533
Instruction BAT Register 3 Upper	634	IBAT3U or SPR534
Instruction BAT Register 3 Lower	635	IBAT3L or SPR535
Data BAT Register 0 Upper	636	DBAT0U or SPR536
Data BAT Register 0 Lower	637	DBAT0L or SPR537
Data BAT Register 1 Upper	638	DBAT1U or SPR538
Data BAT Register 1 Lower	639	DBAT1L or SPR539
Data BAT Register 2 Upper	640	DBAT2U or SPR540
Data BAT Register 2 Lower	641	DBAT2L or SPR541

DWARF Definition

DWARF Release Number

This section defines the Debug With Arbitrary Record Format (DWARF) debugging format for the PowerPC processor family. The PowerPC ABI does not define a debug format. However, all systems that do implement DWARF shall use the following definitions.

DWARF is a specification developed for symbolic, source-level debugging. The debugging information format does not favor the design of any compiler or debugger. For more information on DWARF, see the documents cited in the section **Evolution of the ABI Specification** in Chapter 1.

The DWARF definition requires some machine-specific definitions. The register number mapping needs to be specified for the PowerPC registers. In addition, the DWARF Version 2 specification requires processor-specific address class codes to be defined.

DWARF Register Number Mapping

Table 3-7 outlines the register number mapping for the PowerPC processor family. For kernel debuggers, the mapping for all privileged registers is also defined in Table 3-8. Note that for all special purpose registers, the number is simply 100 plus the SPR register number, as defined in the PowerPC Architecture. Registers with an asterisk before their name are MPC601 chip-specific and are not part of the generic PowerPC chip architecture.

Table 3-7 PowerPC Register Number Mapping

Register Name	Number	Abbreviation
General Register 0-31	`0–31`	`R0–R31`
Floating Register 0-31	`32–63`	`F0–F31`
Condition Register	`64`	`CR`
Floating-Point Status and Control Register	`65`	`FPSCR`
* MQ Register	`100`	`MQ` or `SPR0`
Fixed-Point Exception Register	`101`	`XER` or `SPR1`
* Real Time Clock Upper Register	`104`	`RTCU` or `SPR4`
* Real Time Clock Lower Register	`105`	`RTCL` or `SPR5`
Link Register	`108`	`LR` or `SPR8`
Count Register	`109`	`CTR` or `SPR9`

The above process can be repeated as many times as desired within a single function activation. When it is time to return, the stack pointer is set to the value of the back chain, thereby removing all dynamically allocated stack space along with the rest of the stack frame. Naturally, a program must not reference the dynamically allocated stack area after it has been freed.

Even in the presence of signals, the above dynamic allocation scheme 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 a jump. 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.

The parameter list area holds "overflow" arguments passed in calls to other functions. (See the OTHER label in the algorithm in **Parameter Passing** earlier in this chapter.) Its size is also known to the compiler and can be allocated along with the fixed frame area at function entry. However, the standard calling sequence requires that the parameter list area begin at a fixed offset (8) from the stack pointer, so this area must move when dynamic stack allocation occurs.

Data in the parameter list area are naturally addressed at constant offsets from the stack pointer. However, in the presence of dynamic stack allocation, the offsets from the stack pointer to the data in the local variables area are not constant. To provide addressability, a frame pointer is established to locate the local variables area consistently throughout the function's activation.

Dynamic stack allocation is accomplished by "opening" the stack just above the parameter list area. The following steps show the process in detail:

1. Sometime after a new stack frame is acquired and before the first dynamic space allocation, a new register, the frame pointer, is set to the value of the stack pointer. The frame pointer is used for references to the function's local, non-static variables.
2. The amount of dynamic space to be allocated is rounded up to a multiple of 16 bytes, so that 16-byte stack alignment is maintained.
3. The stack pointer is decreased by the rounded byte count, and the address of the previous stack frame (the back chain) is stored at the word addressed by the new stack pointer. This shall be accomplished atomically by using `stwu rS,-length(r1)` if the length is less than 32768 bytes, or by using `stwux rS,r1,rspace`, where `rS` is the contents of the back chain word and `rspace` contains the (negative) rounded number of bytes to be allocated.

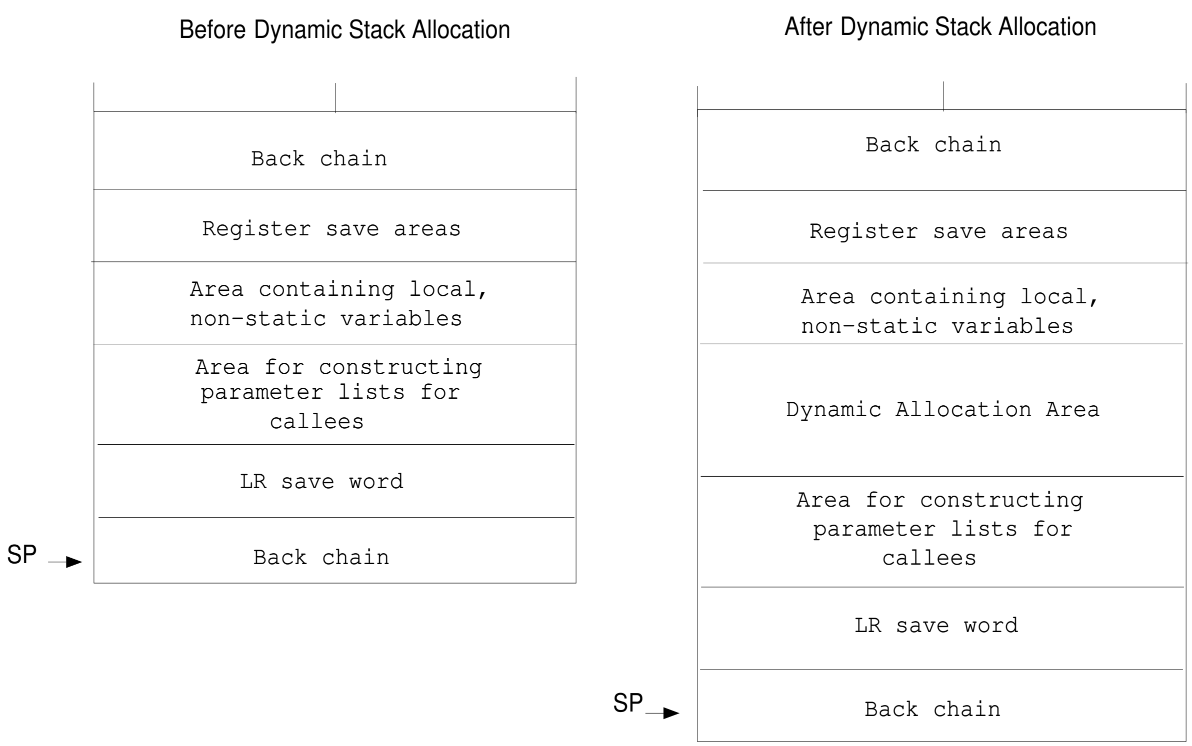

Figure 3-45 Dynamic Stack Space Allocation

C	Assembly

```
switch(j)                                     cmplwi  r12, 4
{                                             bge     .Ldef
case 0:                                       slwi    r12, 2
        ...                                   addis   r12, r12, .Ltab@ha
case 1:                                       lwz     r0, .Ltab@l(r12)
        ...                                   mtctr   r0
case 3:                                       bctr
        ...                                   .rodata
                            .Ltab:            .long    .Lcase0
default:                                      .long    .Lcase1
        ...                                   .long    .Ldef
}                                             .long    .Lcase2
                                              .text
```

Figure 3-43 Absolute `switch` Code

C	Assembly

```
switch(j)                                     cmplwi    r12, 4
{                                             bge       .Ldef
case 0:                                       bl        .L1
        ...                 .L1:              slwi      r12, 2
case 1:                                       mflr      r11
        ...                                   addi      r12, r12, .Ltab-.L1
case 3:                                       add       r0, r12, r11
        ...                                   mtctr     r0
default:                                      bctr
        ...                 .Ltab:            b         .Lcase0
}                                             b         .Lcase1
                                              b         .Ldef
                                              b         .Lcase3
```

Figure 3-44 Position-Independent `switch` Code, All Models

Dynamic Stack Space Allocation

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. The mechanism for allocating dynamic space is embedded completely within a function and does not affect the standard calling sequence. Thus languages that need dynamic stack frame sizes can call C functions, and vice versa.

Figure 3-45 shows the stack frame before and after dynamic stack allocation. The local variables area is used for storage of function data, such as local variables, whose sizes are known to the compiler. This area is allocated at function entry and does not change in size or position during the function's activation.

C **Assembly**

```
extern void func();             .extern func
extern void (*ptr) ();          .extern ptr

                                .text

                                # Assumes GOT pointer in r31

ptr=func;                       addis r11, r31, func@got@ha
                                lwz   r0, func@got@l(r11)
                                addis r12, r31, ptr@got@ha
                                lwz   r12, ptr@got@l(r12)
                                stw   r0, 0(r12)

(*ptr) ();                      addis r12, r31, ptr@got@ha
                                lwz   r12, ptr@got@l(r12)
                                lwz   r0, 0(r12)
                                mtlr  r0
                                blrl
```

Figure 3-41 Large Model Position-Independent Indirect Function Call

Branching

Programs use branch instructions to control their execution flow. As defined by the architecture, branch instructions hold a self-relative value with a 64-Mbyte range, allowing a jump to locations up to 32 Mbytes away in either direction.

C **Assembly**

```
label:                          .L01:
          ...                             ...
          goto label;                     b .L01
```

Figure 3-42 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 `r12`.
- The `case` label constants begin at zero.
- The `case` labels, the `default`, and the address table use assembly names `.Lcase`*i*, `.Ldef`, and `.Ltab`, respectively.

For indirect function calls, a `blrl` instruction is used as shown in Figures 3-39 through 3-41.

C	**Assembly**
`extern void func();`	`.extern func`
`extern void (*ptr) ();`	`.extern ptr`
	`.text`
`ptr = func;`	`addis r6, r0, func@ha`
	`addi  r0, r6, func@l(r6)`
	`addis r7, r0, ptr@ha`
	`stw   r0, ptr@l(r7)`
`(*ptr)();`	
	`addis r6, r0, ptr@ha`
	`lwz   r0, ptr@l(r6)`
	`mtlr  r0`
	`blrl`

Figure 3-39 Absolute Indirect Function Call

C	**Assembly**
`extern void func();`	`.extern func`
`extern void (*ptr) ();`	`.extern ptr`
	`.text`
	`# Assumes GOT pointer in r31`
`ptr = func;`	`lwz  r0,  func@got(r31)`
	`lwz  r12, ptr@got(r31)`
	`stw  r0,  0(r12)`
`(*ptr) ();`	`lwz  r12, ptr@got(r31)`
	`lwz  r0,  0(r12)`
	`mtlr r0`
	`blrl`

Figure 3-40 Small Model Position-Independent Indirect Function Call

C	Assembly

`extern int src;`
`extern int dst;`
`extern int *ptr;`
	`.extern src`
	`.extern dst`
	`.extern ptr`
	
	`.text`
	
	`# Assumes GOT pointer in r31`

`dst = src;`	`addis r6, r31, src@got@ha`
	`lwz   r6, src@got@l(r6)`
	`addis r7, r31, dst@got@ha`
	`lwz   r7, dst@got@l(r7)`
	`lwz   r0, 0(r6)`
	`stw   r0, 0(r7)`

`ptr = &dst;`	`addis r6, r31, dst@got@ha`
	`lwz   r0, dst@got@l(r6)`
	`addis r7, r31, ptr@got@ha`
	`lwz   r7, ptr@got@l(r7)`
	`stw   r0, 0(r7)`

`*ptr = src;`	`addis r6, r31, src@got@ha`
	`lwz   r6, src@got@l(r6)`
	`addis r7, r31, ptr@got@ha`
	`lwz   r7, ptr@got@l(r7)`
	`lwz   r0, 0(r6)`
	`lwz   r7, 0(r7)`
	`stw   r0, 0(r7)`

Figure 3-37 Large Model Position-Independent Load and Store

Function Calls

Programs use the PowerPC `bl` instruction to make direct function calls. A `bl` instruction has a self-relative branch displacement that can reach 32 Mbytes in either direction. Hence, the use of a `bl` instruction to effect a call within an executable or shared object file limits the size of the executable or shared object file text segment.

A compiler normally generates the `bl` instruction to call a function as shown in Figure 3-38. The called function may be in the same module (executable or shared object) as the caller, or it may be in a different module. In the former case, the link editor resolves the symbol and the `bl` branches directly to the called function. In the latter case, the link editor cannot directly resolve the symbol. Instead, it treats the `bl` as a branch to "glue" code that it generates, and the dynamic linker modifies the glue code to branch to the function itself. See **Procedure Linkage Table** in Chapter 5 for more details.

C	Assembly
`extern void func();` `func();`	`.extern func` `bl func`

Figure 3-38 Direct Function Call

Note – In the examples that follow, the assembly syntax *symbol*`@got` refers to the offset in the global offset table at which the value of *symbol* (that is, the address of the variable whose name is *symbol*) is stored, assuming that the offset is no larger than 16 bits. The syntax *symbol*`@got@ha`, *symbol*`@got@h`, and *symbol*`@got@l` refer to the high-adjusted, high, and low parts of that offset, when the offset may be greater than 16 bits.

C	Assembly

`extern int src;`
`extern int dst;`
`extern int *ptr;`
	`.extern src`
	`.extern dst`
	`.extern ptr`
	
	`.text`
	
	`# Assumes GOT pointer in r31`

`dst = src;`	`lwz r6, src@got(r31)`
	`lwz r7, dst@got(r31)`
	`lwz r0, 0(r6)`
	`stw r0, 0(r7)`

`ptr = &dst;`	`lwz r0, dst@got(r31)`
	`lwz r7, ptr@got(r31)`
	`stw r0, 0(r7)`

`*ptr = src;`	`lwz r6, src@got(r31)`
	`lwz r7, ptr@got(r31)`
	`lwz r0, 0(r6)`
	`lwz r7, 0(r7)`
	`stw r0, 0(r7)`

Figure 3-36 Small Model Position-Independent Load and Store

Note – The value of the assembler expression `symbol@l` is the low-order 16 bits of the value of the symbol. The value of the expression `symbol@ha` is the high-order 16 bits of the value of the symbol, adjusted so that when it is shifted left by 16 bits and `symbol@l` is added to it, the resulting value is the value of the symbol. That is, `symbol@ha` compensates as necessary for the carry that may take place because of `symbol@l` being a signed quantity.

Data Objects

This section describes only objects with static storage duration. It excludes stack-resident objects because programs always compute their virtual addresses relative to the stack or frame pointers.

In the PowerPC Architecture, only load and store instructions access memory. Because PowerPC instructions cannot hold 32-bit addresses directly, a program normally computes an address into a register and accesses memory through the register. Symbolic references in absolute code put the symbols' values—or absolute virtual addresses—into instructions.

Position-independent instructions cannot contain absolute addresses. Instead, instructions that reference symbols hold the symbols' (signed) offsets into the global offset table. Combining the offset with the global offset table address in a general register (for example, `r31` loaded in the sample prologue in Figure 3-33) gives the absolute address of the table entry holding the desired address.

Figures 3-35 through 3-37 show sample assembly language equivalents to C language code for absolute and position-independent compilations. It is assumed that all shared objects are compiled position independent and only executable modules may be absolute. The code in the figures contains many redundant operations; it is intended to show how each C statement would have been compiled independently of its context.

C | **Assembly**

```
extern int src;                  .extern src
extern int dst;                  .extern dst
extern int *ptr;                 .extern ptr

dst = src;                       addis r6, r0, src@ha
                                 lwz   r0, src@1(r6)
                                 addis r7, r0, dst@ha
                                 stw   r0, dst@1(r7)

ptr = &dst;                      addis r6, r0, dst@ha
                                 addi  r0, r0, dst@1(r6)
                                 addis r7, r0, ptr@ha
                                 stw   r0, ptr@1(r7)

*ptr = src;                      addis r6, r0, src&ha
                                 lwz   r0, src@1(r6)
                                 addis r7, r0, ptr@ha
                                 lwz   r7, ptr@1(r7)
                                 stw   r0, 0(r7)
```

Figure 3-35 Absolute Load and Store

```
function:     mflr    r0               # Save return address in caller's frame
              stw     r0,4(r1)         # . . .
              ori     r11,r1,0         # Save end of fpr save area
              stwu    r1,-len(r1)      # Establish new frame
              bl      _savefpr_14      # Save float registers
              addi    r11,r11,-144     # Compute end of gpr save area
              bl      _savegpr_14_g    # Save gprs and fetch GOT ptr
              mflr    r31              # Place GOT ptr in r31
                                       # Save CR here if necessary
              ...                      # Body of function
              addi    r11,r1,len-144   # Addr of gpr save area to r11
              bl      _restgpr_14      # Restore gprs
                                       # Restore CR here if necessary
              addi    r11,r11,144      # Compute end of frame/fprs
              bl      _restfpr_14_x    # Restore fprs and return
```

Figure 3-33 Prologue and Epilogue Sample Code

Profiling

This section shows a way of providing profiling (entry counting) on PowerPC systems. An ABI-conforming system is not required to provide profiling; however if it does, this is one possible (not required) implementation.

If a function is to be profiled, it saves the link register in the `LR` save word of its caller's stack frame, loads into `r0` a pointer to a word-aligned, one-word, static data area initialized to zero in which the `_mcount` routine is to maintain a count of the number of entries, and calls `_mcount`. For example, the code in Figure 3-34 can be inserted at the beginning of a function, before any other prologue code. The `_mcount` routine is required to restore the link register from the stack so that the profiling code can be inserted transparently, whether or not the profiled function saves the link register itself.

```
.function_mc:            .data
                         .align       2
                         .long        0
                         .text

function:                mflr         r0
                         addis        r11,r0,.funtion_mc@ha
                         stw          r0,4(r1)
                         addi         r0,r11,.function_mc@1
                         bl           _mcount
```

Figure 3-34 Code for Profiling

There are three families of register restoring functions:

- The "simple" register restoring functions, `_rest[fg]pr_`*n*, restore the indicated registers and return.
- The "exit" functions, `_rest[fg]pr_`*n*`_x`, restore the indicated registers and, relying on the registers being restored to be adjacent to the back chain word, restore the link register from the `LR` save word, remove the stack frame, and return through the link register.
- The "tail" functions, `_rest[fg]pr_`*n*`_t`, restore the registers, place the `LR` save word into `r0`, remove the stack frame, and return to their caller. The caller can then implement a "tail call" by moving `r0` into the link register and branching to the tail function. The tail function then sees an apparent call from the function above the one that made the tail call and, when done, returns directly to it.

Figure 3-32 shows an implementation of the `_restfpr_`*n*`_x` family of functions.

```
_restfpr_14_x:  lfd      r14, -144(r11)
_restfpr_15_x:  lfd      r15, -136(r11)
_restfpr_16_x:  lfd      r16, -128(r11)
_restfpr_17_x:  lfd      r17, -120(r11)
_restfpr_18_x:  lfd      r18, -112(r11)
_restfpr_19_x:  lfd      r19, -104(r11)
_restfpr_20_x:  lfd      r20, -96(r11)
_restfpr_21_x:  lfd      r21, -88(r11)
_restfpr_22_x:  lfd      r22, -80(r11)
_restfpr_23_x:  lfd      r23, -72(r11)
_restfpr_24_x:  lfd      r24, -64(r11)
_restfpr_25_x:  lfd      r25, -56(r11)
_restfpr_26_x:  lfd      r26, -48(r11)
_restfpr_27_x:  lfd      r27, -40(r11)
_restfpr_28_x:  lfd      r28, -32(r11)
_restfpr_29_x:  lfd      r29, -24(r11)
_restfpr_30_x:  lfd      r30, -16(r11)
_restfpr_31_x:  lwz      r0, 4(r11)
                lfd      r31, -8(r11)
                mtlr     r0
                ori      r1, r11, 0
                blr
```

Figure 3-32 `_restfpr_`*n*`_x` Implementation

Figure 3-33 below shows sample prologue and epilogue code with full saves of all the nonvolatile floating-point and general registers and a stack frame size of less than 32 Kbytes. The example assumes that the function does not alter the nonvolatile fields of the `CR` and does no dynamic stack allocation.

Note – This code assumes that the size of the module (executable or shared object) in which the code appears is such that a relative branch is able to reach from any part of the text section to any part of the global offset table (or the procedure linkage table, discussed in Chapter 5). Since relative branches can reach +/- 32 Mbytes, this is not considered a serious restriction.

In-line code may be used to save or restore nonvolatile general or floating-point registers that the function uses. However, if there are many registers to be saved or restored, it may be more efficient to call one of the system subroutines described below.

Note that "Load and Store Multiple" PowerPC instructions should not be used on Little-Endian implementations because they cause alignment exceptions, or on Big-Endian implementations because they are slower than the register-at-a-time saves.

If any of the nonvolatile fields of the Condition Register (`CR`) are used, they must also be preserved and restored.

A function that is position independent will probably want to load a pointer to the global offset table into a nonvolatile register. This may be omitted if the function makes no external data references. If external data references are only made within conditional code, loading the global offset table pointer may be deferred until it is known to be needed.

Register Saving and Restoring Functions

The register saving and restoring functions described in this section use nonstandard calling conventions which require them to be statically linked into any executable or shared object modules in which they are used. Thus their interfaces are private, within module interfaces, and therefore are not part of the ABI. They are defined here only to encourage uniformity among compilers in the code used to save and restore registers.

On entry, all the functions described in this section expect `r11` to contain the address of the word just beyond the end of the floating-point or general register save area, as appropriate, and they leave `r11` undisturbed. For example (assuming a stack frame as described in Figure 3-25), on entry to the floating-point register saving and restoring functions, `r11` contains the address of the back chain word of the previous frame (the word just beyond the floating-point register save area). Similarly, on entry to the general register saving and restoring functions, `r11` contains either the address of the first word of the floating-point register save area or, if there is no floating-point register save area, the address of the back chain word. Higher-numbered registers are saved at higher addresses within a save area.

The saving and restoring functions save and restore consecutive general or floating-point registers from register 31 through register *n*, with *n* being between 14 and 31. That is, each "function" described in this section is a family of 36 functions with identical behavior except for the number and kind of registers affected. The function names below use the notation "`[fg]`" to designate the use of an "f" for the floating register functions and a "g" for the general register functions.

There are two families of register saving functions:

- The "simple" register saving functions, `_save[fg]pr_`*n*, save the indicated registers and return.
- The "GOT" register saving functions, `_save[fg]pr_`*n*`_g`, do not return directly. Instead they branch to `_GLOBAL_OFFSET_TABLE_-4`, relying on a `blrl` instruction at that address (see the section **Global Offset Table** in Chapter 5) to return to the caller of the save function with the address of the global offset table in the link register.

Because the PowerPC Architecture provides EA-relative branch instructions and also branch instructions using registers that hold the transfer address, compilers can satisfy the first condition easily.

A "Global Offset Table," or GOT, provides information for address calculation. Position-independent object files (executable and shared object files) have a table in their data segment that holds addresses. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual address 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.

Two position-independent models give programs a choice between more efficient code with some size restrictions and less efficient code without those restrictions. Because of the processor's architecture, a global offset table with no more than 16384 entries (65536 bytes) is more efficient than a larger one. Programs that need more entries must use the larger, more general code. In the following sections, the term "small model" position-independent code is used to refer to code that assumes the smaller global offset table, and "large model" position-independent code is used to refer to the general code.

Function Prologue and Epilogue

This section describes functions' prologue and epilogue code. A function's prologue establishes a stack frame, if necessary, and may save any nonvolatile registers it uses. A function's epilogue generally restores registers that were saved in the prologue code, restores the previous stack frame, and returns to the caller.

Except for the rules below, this ABI does not mandate predetermined code sequences for function prologues and epilogues. However, the following rules, which permit reliable call chain backtracing, shall be followed:

1. Before a function calls any other function, it shall establish its own stack frame, whose size shall be a multiple of 16 bytes, and shall save the link register at the time of entry in the `LR` save word of its caller's frame.
2. If a function establishes a stack frame, it shall update the back chain word of the stack frame atomically with the stack pointer (`r1`) using one of the "Store Word with Update" instructions.
   - For small (no larger than 32 Kbytes) stack frames, this may be accomplished with a "Store Word with Update" instruction with an appropriate negative displacement.
   - For larger stack frames, the prologue shall load a volatile register with the two's complement of the size of the frame (computed with `addis` and `addi` or `ori` instructions) and issue a "Store Word with Update Indexed" instruction.
3. The only permitted references with negative offsets from the stack pointer are those described here for establishing a stack frame.
4. When a function deallocates its stack frame, it must do so atomically, either by loading the stack pointer (`r1`) with the value in the back chain field or by incrementing the stack pointer by the same amount by which it has been decremented.

Coding Examples

This section describes 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 discussed how a program may use the machine or the operating system, and they specified what a program may and may not assume about the execution environment. Unlike previous material, the information in this section 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' 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.

The following sections describe the differences between these models. When different, code sequences for the models appear together for easier comparison.

Note – The examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences or to reproduce compiler output. None of them reference data in the small data area.

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 if the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques:

- Control transfer instructions hold addresses relative to the Effective Address (EA) or use registers that hold the transfer address. An EA-relative branch computes its destination address in terms of the current EA, *not* relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in instructions (in the text segment), the compiler generates code to calculate an absolute address (in a register or in the stack or data segment) during execution.

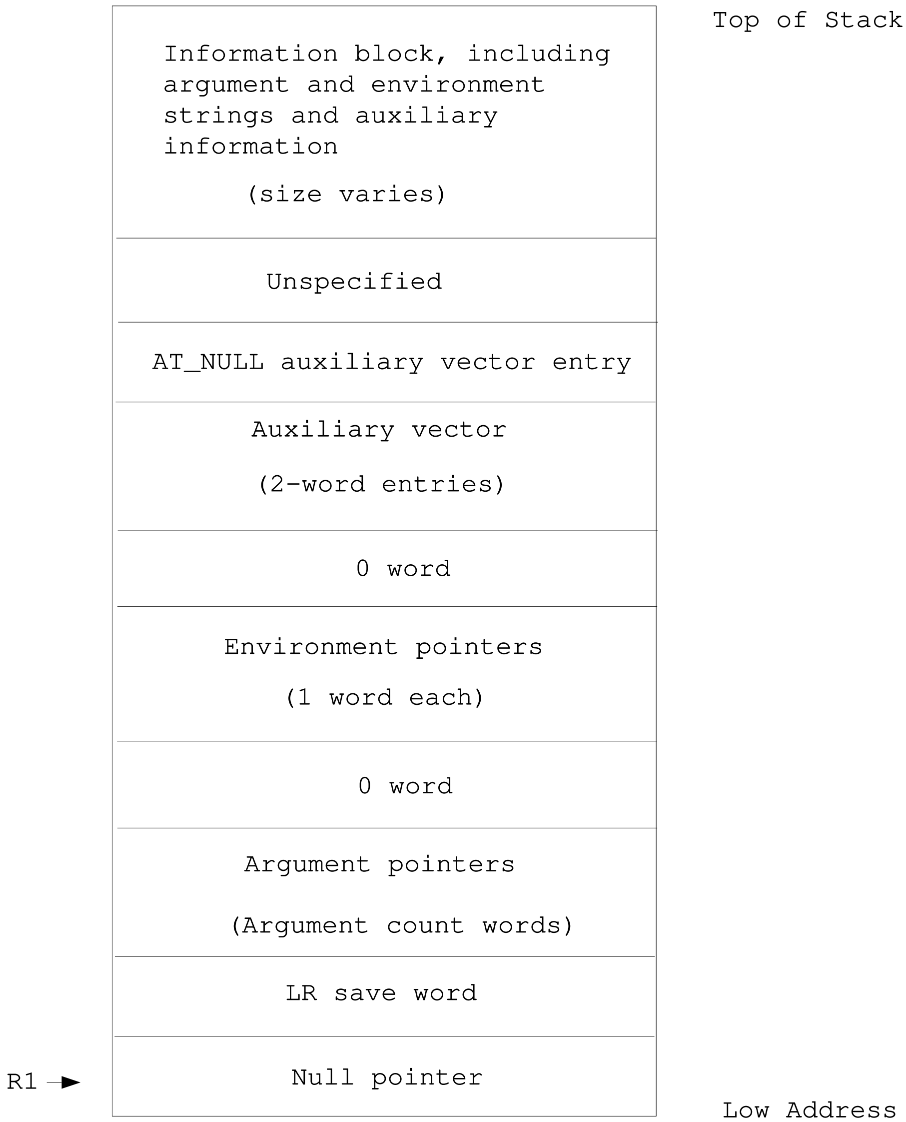

Figure 3-31 Initial Process Stack

AT_DCACHEBSIZE — The a_val member of this entry gives the data cache block size for processors on the system on which this program is running. If the processors have unified caches, AT_DCACHEBSIZE is the same as AT_UCACHEBSIZE.

AT_ICACHEBSIZE — The a_val member of this entry gives the instruction cache block size for processors on the system on which this program is running. If the processors have unified caches, AT_DCACHEBSIZE is the same as AT_UCACHEBSIZE.

AT_UCACHEBSIZE — The a_val member of this entry is zero if the processors on the system on which this program is running do not have a unified instruction and data cache. Otherwise, it gives the cache block size.

Other auxiliary vector types are reserved. No flags are currently defined for AT_FLAGS on the PowerPC Architecture.

When a process receives control, its stack holds the arguments, environment, and auxiliary vector from exec(BA_OS). Argument strings, environment strings, and the auxiliary information appear in no specific order within the information block; the system makes no guarantees about their relative arrangement. The system may also leave an unspecified amount of memory between the null auxiliary vector entry and the beginning of the information block. The back chain word of the first stack frame contains a null pointer (0). A sample initial stack is shown in Figure 3-31.

a_type auxiliary vector types are described below.

Name	Description
AT_NULL	The auxiliary vector has no fixed length; instead an entry of this type denotes the end of the vector. The corresponding value of a_un is undefined.
AT_IGNORE	This type indicates the entry has no meaning. The corresponding value of a_un is undefined.
AT_EXECFD	As Chapter 5 in the *System V ABI* 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 an 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 the section **Program Header** in Chapter 5 of the *System V ABI* and the section **Program Loading** in Chapter 5 of this 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 is also 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 the section **Program Header** in Chapter 5 of the *System V ABI* for more information about the base address.
AT_FLAGS	If present, the a_val member of this entry holds 1-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.

Process Stack

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 general purpose register `r1`. 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 structures, which are defined in Figure 3-30.

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

Figure 3-30 Auxiliary Vector Structure

The structures are interpreted according to the `a_type` member, as shown in Table 3-6.

Table 3-6 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_DCACHEBSIZE	10	a_val
AT_ICACHEBSIZE	11	a_val
AT_UCACHEBSIZE	12	a_val

Process Initialization

This section describes the machine state that `exec(BA_OS)` creates for "infant" processes, including argument passing, register usage, and stack frame layout. Programming language systems use this initial program state to establish a standard environment for their application programs. For example, a C program begins executing at a function named `main`, conventionally declared in the way described in Figure 3-29.

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

Figure 3-29 Declaration for `main`

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.

Registers

When a process is first entered (from an `exec(BA_OS)` system call), the contents of registers other than those listed below are unspecified. 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. Following are the registers whose contents are specified:

`r1` The initial stack pointer, aligned to a 16-byte boundary and pointing to a word containing a NULL pointer.

`r3` Contains `argc`, the number of arguments.

`r4` Contains `argv`, a pointer to the array of argument pointers in the stack. The array is immediately followed by a NULL pointer. If there are no arguments, `r4` points to a NULL pointer.

`r5` Contains `envp`, a pointer to the array of environment pointers in the stack. The array is immediately followed by a NULL pointer. If no environment exists, `r5` points to a NULL pointer.

`r6` Contains a pointer to the auxiliary vector. The auxiliary vector shall have at least one member, a terminating entry with an `a_type` of `AT_NULL` (see Figure 3-30 and Table 3-6).

`r7` Contains a termination function pointer. If `r7` contains a nonzero value, the value represents a function pointer that the application should register with `atexit(BA_OS)`. If `r7` contains zero, no action is required.

`fpscr` Contains 0, specifying "round to nearest" mode, IEEE Mode, and the disabling of floating-point exceptions.

Due to the pipelined nature of the PowerPC, more than one instruction may be executing concurrently. When an exception occurs, all unexecuted instructions that appear earlier in the instruction stream are allowed to complete. As a result of completing these instructions, additional exceptions may be generated. All such exceptions are handled in order.

The operating system partitions the set of concurrent exceptions into subsets, all of whose exceptions share the same signal number. Each subset of exceptions is delivered as a single signal. The multiple signals resulting from multiple concurrent exceptions are delivered in unspecified order.

Exception Interface

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, 1) information (such as the address of the instruction that should be executed after control is returned to the original program and the contents of the machine state register) is saved, 2) program control passes from user to supervisor level, and 3) software continues execution at an address (exception vector) predetermined for each exception.

Exceptions may be synchronous or asynchronous. Synchronous exceptions, being caused by instruction execution, can be explicitly generated by a process. The operating system handles an exception either by completing the faulting operation in a manner transparent to the application or by delivering a signal to the application. The correspondence between exceptions and signals is shown in Table 3-5.

Table 3-5 Exceptions and Signals

Exception Name	Signal	Examples
Illegal instruction	`SIGILL`	Illegal or privileged instruction, invalid instruction form Optional, unimplemented instruction
Storage access	`SIGSEGV`	Unmapped instruction or data location access Storage protection violation
Alignment	`SIGBUS`	Invalid data item alignment Execution of a string or load/store multiple instruction in Little-Endian mode
Trap instruction	`SIGTRAP`	Execution of `tw` instruction (see **Note** below)
Floating unavailable	`SIGFPE`	Floating instruction is not implemented
Floating exception	`SIGFPE`	Floating-point overflow or underflow Floating-point divide by zero Floating-point conversion overflow Other enabled floating-point exceptions

---

Note – The `tw` instructions with all five condition bits set are reserved for system use (for example, breakpoint implementation), so applications should not rely on the behavior of such traps.

The signals that an exception may give rise to are `SIGILL`, `SIGSEGV`, `SIGBUS`, `SIGTRAP`, and `SIGFPE`. If one of these signals is generated due to an exception when the signal is blocked, the behavior is undefined.

Managing the Process Stack

The 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 a single system. A program, therefore, should not depend on finding its stack at a particular virtual address.

A tunable configuration parameter controls the system maximum stack size. A process can also use `setrlimit(BA_OS)` to set its own maximum stack size, up to the system limit. The stack segment is both readable and writable.

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 request the system to use an address the program supplies. The 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 segments 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. Thus, a program that used `mmap(KE_OS)` to request a mapping at a specific address could appear to work in some environments and fail in others. For this reason, programs that want to establish a mapping in their address space should let the system choose the address.

Despite these warnings about requesting specific addresses, the facility can be used properly. 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

Two execution modes exist in the PowerPC Architecture: user and supervisor. Processes run in user mode (the less privileged). The operating system kernel runs in supervisor mode. A program executes an `sc` instruction to change execution modes.

Note that the ABI does not define the implementation of individual system calls. Instead, programs shall use the system libraries described in Chapter 6. Programs with embedded system call or trap instructions do not conform to the ABI.

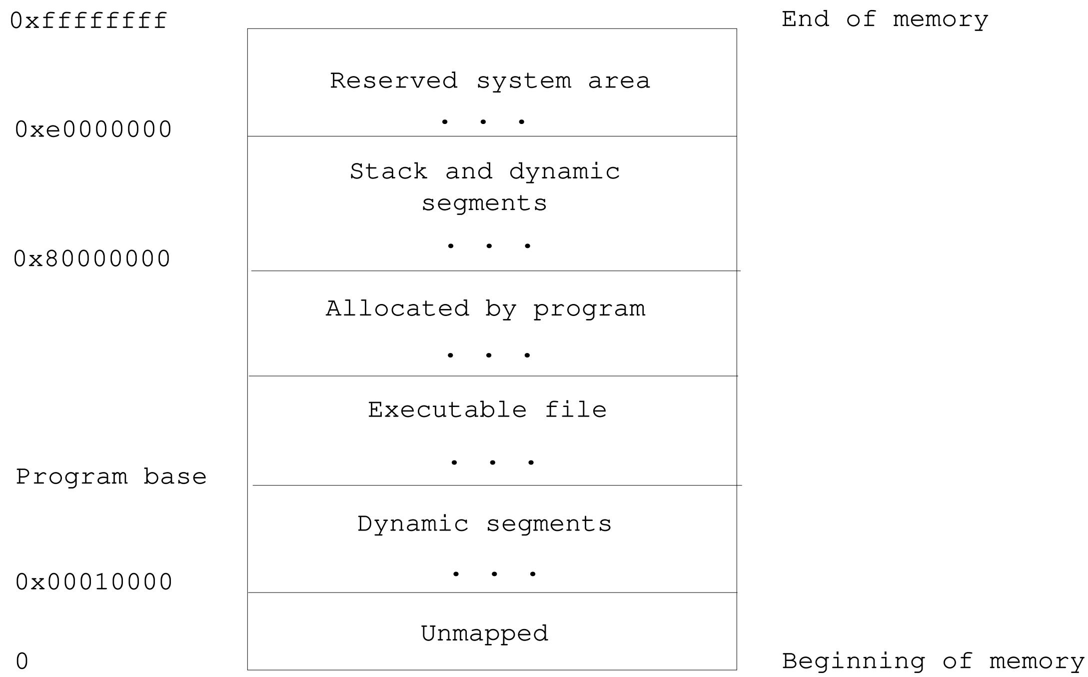

Figure 3-28 Virtual Address Configuration

Note – Although application programs may begin at virtual address 0, they conventionally begin above `0x10000` (64 Kbytes), leaving the initial 64 Kbytes with an invalid address mapping. Processes that reference this invalid memory (for example, by dereferencing a null pointer) generate an access exception trap, as described in the section **Exception Interface** in this chapter.

Note – A program base of `0x02000000` (32 Mbytes) is recommended, for reasons given in Chapter 5.

As Figure 3-28 shows, the system reserves the high end of virtual space, with a process' stack and 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 512 Mbytes from the virtual address space. Thus, the user virtual address range has a minimum upper bound of `0xdfffffff`. Individual systems may reserve less space, increasing the process virtual memory range. More information follows in the next section, **Managing the Process Stack**.

Although applications may control their memory assignments, the typical arrangement follows the diagram above. When applications let the system choose addresses for dynamic segments (including shared object segments), it will prefer addresses below the beginning of the executable and above 64 Kbytes, or addresses above 2 Gbytes. This leaves the "middle" of the address spectrum, those addresses above the executable and below 2 Gbytes, available for dynamic memory allocation with facilities such as `malloc(BA_OS)`.

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." An object file may contain more segments (for example, for debugger use), and a process can also create additional segments for itself with system services.

Note – The term "virtual address" as used in this document refers to a 32-bit address generated by a program, as contrasted with the physical address to which it is mapped. The PowerPC Architecture documentation refers to this type of address as an "effective address."

Page Size

Memory is organized into pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another. Processes may call `sysconf(BA_OS)` to determine the system's current page size. Currently, the only valid hardware page size for the PowerPC Architecture is 4096 bytes (4 Kbytes), but this ABI allows the underlying operating system to cluster pages into logical power-of-two page sizes up to 65536 bytes (64 Kbytes).

Virtual Address Assignments

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

- The system reserves a configuration-dependent amount of virtual space.
- A tunable configuration parameter limits process size.
- 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.

Figure 3-28 shows the virtual address configuration on the PowerPC Architecture. The segments with different properties are typically grouped in different areas of the address space. A reserved area resides at the top of the virtual space and is used by the system. The loadable segments may begin at zero (0); the exact addresses depend on the executable file format (see Chapters 4 and 5). The process' stack and dynamic segments reside below the system-reserved area. Processes can control the amount of virtual memory allotted for stack space, as described below.

containing a trailing ellipsis mark (`...`), but compiler vendors are expected to provide options for non-ANSI programs to allow them to declare variable argument functions in the command line or to treat all non-prototyped functions as (potentially) having variable argument lists.

Return Values

Functions shall return `float` or `double` values in `f1`, with `float` values rounded to single precision. Functions shall return values of type `int`, `long`, `enum`, `short`, and `char`, or a pointer to any type as unsigned or signed integers as appropriate, zero- or sign-extended to 32 bits if necessary, in `r3`. A structure or union whose size is less than or equal to 8 bytes shall be returned in `r3` and `r4`, as if it were first stored in an 8-byte aligned memory area and then the low-addressed word were loaded into `r3` and the high-addressed word into `r4`. Bits beyond the last member of the structure or union are not defined.

Values of type `long long` and `unsigned long long`, where supported, shall be returned with the lower addressed word in `r3` and the higher in `r4`.

Values of type `long double` and structures or unions that do not meet the requirements for being returned in registers are returned in a storage buffer allocated by the caller. The address of this buffer is passed as a hidden argument in `r3` as if it were the first argument, causing `gr` in the argument passing algorithm above to be initialized to 4 instead of 3.

Table 3-4 Parameter Passing Example Register Allocation

General Purpose Registers	Floating-Point Registers	Stack Frame Offset
`r3: c`	`f1: ff`	`08: ptr to t`
`r4: d`	`f2: gg`	`0c: (padding)`
`r5: e`	`f3: hh`	`10: nn(lo)`
`r6: f`	`f4: ii`	`14: nn(hi)`
`r7: g`	`f5: jj`	
`r8: h`	`f6: kk`	
`r9: ptr to ld`	`f7: ll`	
`r10: ptr to s`	`f8: mm`	

Note – In Table 3-4, (`lo`) and (`hi`) denote the low- and high-addressed word of the `double` value as stored in memory, regardless of the Endian mode of the implementation. The `ptr to` arguments are pointers to copies if necessary to preserve call-by-value semantics.

Variable Argument Lists

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments are passed 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 implementations. However, they do not work on the PowerPC Architecture because some arguments are passed in registers. Portable C programs use the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists on PowerPC and other machines as well.

A caller of a function that takes a variable argument list shall set condition register bit 6 to 1 if it passes one or more arguments in the floating-point registers. It is strongly recommended that the caller set the bit to 0 otherwise, using the `creqv 6, 6, 6` (set to 1) or `crxor 6, 6, 6` (set to 0) instruction.

The motivation for using the condition register bit is twofold. First, a function that takes a variable argument list may test condition register bit 6 to determine whether or not to store the floating-point argument registers in memory, thereby making execution of such functions more efficient when there are no floating-point arguments. Second, programs that do not otherwise use floating point need not acquire a floating-point state, with the attendant saving and restoring of the state on context switches, merely because they call functions with variable argument lists. The cost for these savings is one additional, non-memory-reference instruction in the callers of functions that accept variable argument lists. ANSI C requires that such functions be declared with a prototype

If `gr>10`, go to OTHER. Otherwise, load the argument value into general register `gr`, set `gr` to `gr+1`, and go to SCAN. Values shorter than 32 bits are sign-extended or zero-extended, depending on whether they are signed or unsigned.

`LONG_LONG`:

Note that implementations are not required to support a `long long` data type, but if they do, the following treatment is required.

If `gr>9`, go to OTHER. If `gr` is even, set `gr` to `gr+1`. Load the lower-addressed word of the `long long` into `gr` and the higher-addressed word into `gr+1`, set `gr` to `gr+2`, and go to SCAN.

OTHER:

Arguments not otherwise handled above are passed in the parameter words of the caller's stack frame. `SIMPLE_ARG`s, as defined above, are considered to have 4-byte `size` and alignment, with simple integer types shorter than 32 bits sign- or zero-extended (conceptually) to 32 bits. `float`, `long long` (where implemented), and `double` arguments are considered to have 8-byte `size` and alignment, with `float` arguments converted to `double` representation. Round `starg` up to a multiple of the alignment requirement of the argument and copy the argument byte-for-byte, beginning with its lowest addressed byte, into `starg, ..., starg+size−1`. Set `starg` to `starg+size`, then go to SCAN.

The contents of registers and words skipped by the above algorithm for alignment (padding) are undefined.

As an example, assume the declarations and the function call shown in Figure 3-27. The corresponding register allocation and storage would be as shown in Table 3-4.

```
typedef struct {
        int a, b;
        double  dd;          /* doubleword aligned */
} sparm;
sparm   s, t;
int     c, d, e, f, g, h;
long double ld;
double  ff, gg, hh, ii, jj, kk, ll, mm, nn;

x = func(c, ff, d, gg, e, hh, f, ii, g, jj, h, ld, kk, ll, s, mm, t, nn);
```

Figure 3-27 Parameter Passing Example

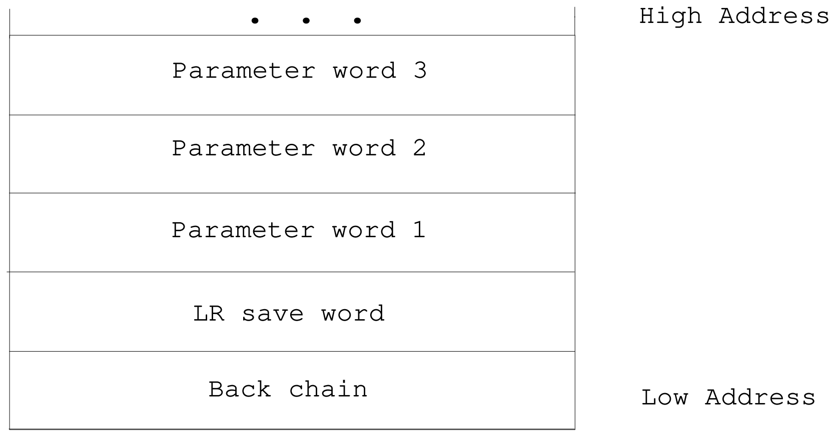

Figure 3-26 Parameter List Area

The following algorithm specifies where argument data is passed for the C language. For this purpose, consider the arguments as ordered from left (first argument) to right, although the order of evaluation of the arguments is unspecified. In this algorithm, `fr` contains the number of the next available floating-point register, `gr` contains the number of the next available general purpose register, and `starg` is the address of the next available stack argument word.

INITIALIZE:

Set `fr=1`, `gr=3`, and `starg` to the address of parameter word 1.

SCAN:

If there are no more arguments, terminate. Otherwise, select one of the following depending on the type of the next argument:

`DOUBLE_OR_FLOAT`:

If `fr>8` (that is, there are no more available floating-point registers), go to OTHER. Otherwise, load the argument value into floating-point register `fr`, set `fr` to `fr+1`, and go to SCAN.

`SIMPLE_ARG`:

A `SIMPLE_ARG` is one of the following:

- One of the simple integer types no more than 32 bits wide (`char`, `short`, `int`, `long`, `enum`), or
- A pointer to an object of any type, or
- A `struct`, `union`, or `long double`, any of which shall be treated as a pointer to the object, or to a copy of the object where necessary to enforce call-by-value semantics. Only if the caller can ascertain that the object is "constant" can it pass a pointer to the object itself.

- Before a function changes the value in any nonvolatile general register, `r`*n*, it shall save the value in `r`*n* in the word in the general register save area 4*(32-*n*) bytes before the low-addressed end of the floating-point register save area.
- Before a function changes the value in any nonvolatile field in the condition register, it shall save the values in all the nonvolatile fields of the condition register at the time of entry to the function in the `CR` save area.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size. The minimum stack frame consists of the first two words, described below, with padding to the required 16-byte alignment. The calling sequence also does not restrict how a language system uses the "local variable space" of the standard stack frame or how large it should be.

The stack frame header consists of the back chain word and the `LR` save word. The back chain word always contains a pointer to the previously allocated stack frame. Before a function calls another function, it shall save the contents of the link register at the time the function was entered in the `LR` save word of its caller's stack frame and shall establish its own stack frame.

Except for the stack frame header and any padding necessary to make the entire frame a multiple of 16 bytes in length, a function need not allocate space for the areas that it does not use. If a function does not call any other functions and does not require any of the other parts of the stack frame, it need not establish a stack frame. Any padding of the frame as a whole shall be within the local variable area; the parameter list area shall immediately follow the stack frame header, and the register save areas shall contain no padding.

Parameter Passing

For a RISC machine such as PowerPC, it is generally more efficient to pass arguments to called functions in registers (both general and floating-point registers) than to construct an argument list in storage or to push them onto a stack. Since all computations must be performed in registers anyway, memory traffic can be eliminated if the caller can compute arguments into registers and pass them in the same registers to the called function, where the called function can then use them for further computation in the same registers. The number of registers implemented in a processor architecture naturally limits the number of arguments that can be passed in this manner.

For PowerPC, up to eight words are passed in general purpose registers, loaded sequentially into general purpose registers `r3` through `r10`. In addition, up to eight floating-point arguments can be passed in floating-point registers `f1` through `f8`. If fewer (or no) arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

Only when the "worst-case" arguments passed from a function will not fit in the eight general purpose registers and the eight floating-point registers provided must a function allocate space for arguments in its stack frame; in that case, it needs to allocate only enough space to hold the arguments that do not fit into registers.

The Stack Frame

In addition to the registers, each function may have a stack frame on the runtime stack. This stack grows downward from high addresses. Figure 3-25 shows the stack frame organization. `SP` in the figure denotes the stack pointer (general purpose register `r1`) of the called function after it has executed code establishing its stack frame.

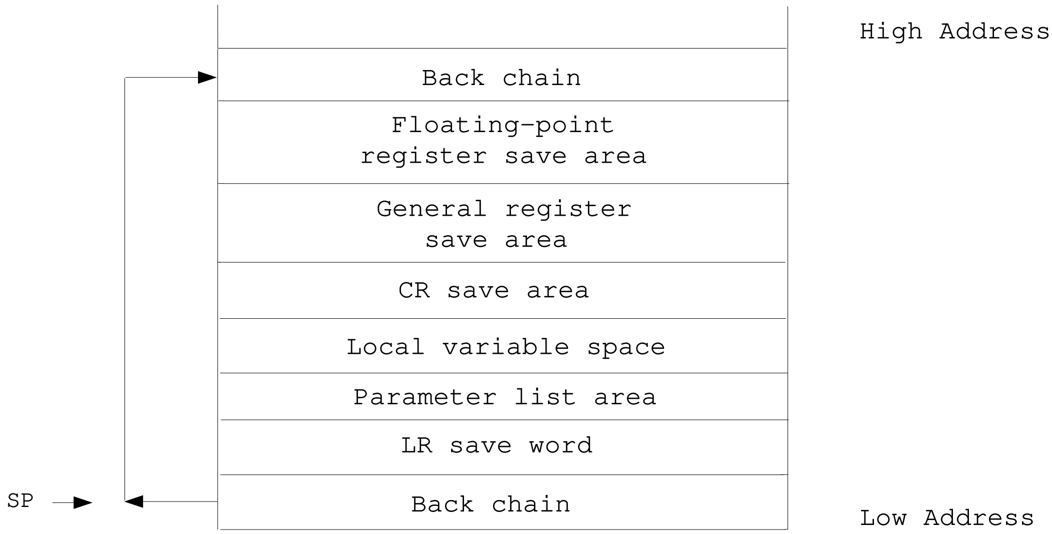

Figure 3-25 Standard Stack Frame

The following requirements apply to the stack frame:

- The stack pointer shall maintain 16-byte alignment.
- The stack pointer shall point to the first word of the lowest allocated stack frame, the "back chain" word. The stack shall grow downward, that is, toward lower addresses. The first word of the stack frame shall always point to the previously allocated stack frame (toward higher addresses), except for the first stack frame, which shall have a back chain of 0 (NULL).
- The stack pointer shall be decremented by the called function in its prologue, if required, and restored prior to return.
- The stack pointer shall be decremented and the back chain updated atomically using one of the "Store Word with Update" instructions, so that the stack pointer always points to the beginning of a linked list of stack frames.
- The parameter list area shall be allocated by the caller and shall be large enough to contain the arguments that the caller stores in it. Its contents are not preserved across calls.
- The sizes of the floating-point and general register save areas may vary within a function and are as determined by the "tags" described in **Special Sections** in Chapter 4.
- Before a function changes the value in any nonvolatile floating-point register, `fr`*n*, it shall save the value in `fr`*n* in the double word in the floating-point register save area 8*(32-*n*) bytes before the back chain word of the previous frame.

The following registers have assigned roles in the standard calling sequence:

`r1`	The stack pointer (stored in `r1`) shall maintain 16-byte alignment. It shall always point to the lowest allocated, valid stack frame, and grow toward low addresses. The contents of the word at that address always point to the previously allocated stack frame. If required, it can be decremented by the called function; see **Dynamic Stack Space Allocation** later in this chapter.
`r3` through `r10` and `f1` through `f8`	These sets of volatile registers may be modified across function invocations and shall therefore be presumed by the calling function to be destroyed. They are used for passing parameters to the called function; see **Parameter Passing** in this chapter. In addition, registers `r3`, `r4`, and `f1` are used to return values from the called function, as described in **Return Values**.
`CR` bit 6 (`CR1`, "floating-point invalid exception")	This bit shall be set by the caller of a "variable argument list" function, as described in **Variable Argument Lists** later in this chapter.
`LR` (Link Register)	This register shall contain the address to which a called function normally returns. `LR` is volatile across function calls.

Signals can interrupt processes (see `signal (BA_OS)` in the *System V Interface Definition*). 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 all registers restored to their original values. Thus, programs and compilers may freely use all registers above except those reserved for system use without the danger of signal handlers inadvertently changing their values.

Table 3-3 Processor Registers *(Continued)*

Register Name	Usage
f9–f13	Volatile registers
f14–f31	Registers used for local variables
CR0–CR7	Condition Register Fields, each 4 bits wide
LR	Link Register
CTR	Count Register
XER	Fixed-Point Exception Register
FPSCR	Floating-Point Status and Control Register

Registers r1, r14 through r31, and f14 through f31 are nonvolatile; that is, they "belong" to the calling function. A called function shall save these registers' values before it changes them, restoring their values before it returns. Registers r0, r3 through r12, f0 through f13, and the special purpose registers CTR and XER are volatile; that is, they are not preserved across function calls. Furthermore, the values in registers r0, r11, and r12 may be altered by cross-module calls, so a function cannot depend on the values in these registers having the same values that were placed in them by the caller.

Register r2 is reserved for system use and should not be changed by application code.

Register r13 is the small data area pointer. Process startup code for executables that reference data in the small data area with 16-bit offset addressing relative to r13 must load the base of the small data area (the value of the loader-defined symbol _SDA_BASE_) into r13. Shared objects shall not alter the value in r13. See **Small Data Area** in Chapter 4 for more details.

Languages that require "environment pointers" shall use r31 for that purpose.

Fields CR2, CR3, and CR4 of the condition register are nonvolatile (value on entry must be preserved on exit); the rest are volatile (value in the field need not be preserved). The VE, OE, UE, ZE, XE, NI, and RN (rounding mode) bits of the FPSCR may be changed only by a called function (for example, fpsetround()) that has the documented effect of changing them. The rest of the FPSCR is volatile.

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, and parameter passing. 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 as long as they conform to the tag requirements for stack traceback; see **Stack Traceback Using Tags** in Chapter 4. Nonetheless, it is recommended that all functions use the standard calling sequences 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

The PowerPC Architecture provides 32 general purpose registers, each 32 bits wide. In addition, the architecture provides 32 floating-point registers, each 64 bits wide, and several special purpose registers. All of the integer, special purpose, and floating-point registers are global to all functions in a running program. Brief register descriptions appear in Table 3-3, followed by more detailed information about the registers.

Table 3-3 Processor Registers

Register Name	Usage
`r0`	Volatile register which may be modified during function linkage
`r1`	Stack frame pointer, always valid
`r2`	System-reserved register
`r3-r4`	Volatile registers used for parameter passing and return values
`r5-r10`	Volatile registers used for parameter passing
`r11-r12`	Volatile registers which may be modified during function linkage
`r13`	Small data area pointer register
`r14-r30`	Registers used for local variables
`r31`	Used for local variables or "environment pointers"
`f0`	Volatile register
`f1`	Volatile register used for parameter passing and return values
`f2-f8`	Volatile registers used for parameter passing

Note – In Figures 3-23 and 3-24, the presence of the unnamed `int` and `short` fields do not affect the alignment of the structure. They align the named members relative to the beginning of the structure, but the named members may not be aligned in memory on suitable boundaries. For example, the `d` members in an array of these structures will not all be on an `int` (4-byte) boundary.

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

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

Halfword aligned, `sizeof` is 2

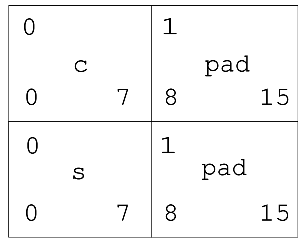

Figure 3-22 `union` Allocation—Big-Endian

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

Byte aligned, `sizeof` is 9

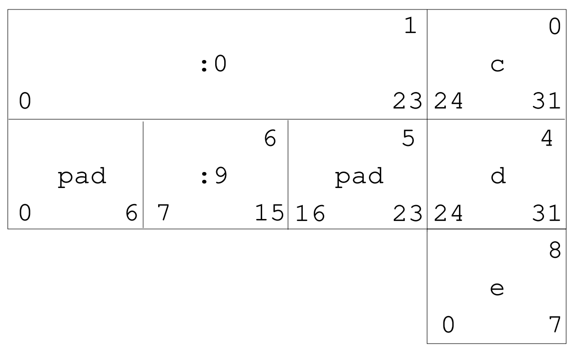

Figure 3-23 Unnamed Bit-Fields—Little-Endian

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

Byte aligned, `sizeof` is 9

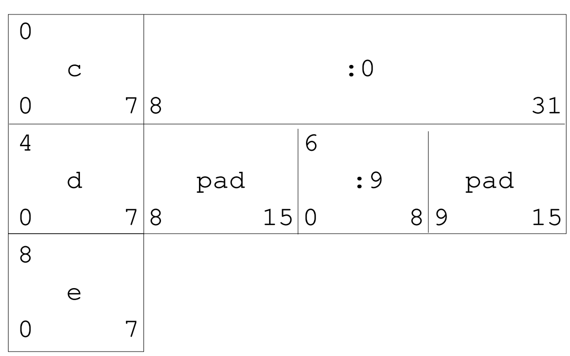

Figure 3-24 Unnamed Bit-Fields—Big-Endian

Halfword aligned, `sizeof` is 2

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

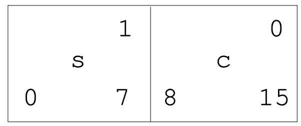

Figure 3-19 Storage Unit Sharing—Little-Endian

Halfword aligned, `sizeof` is 2

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

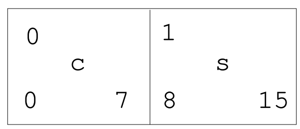

Figure 3-20 Storage Unit Sharing—Big-Endian

Halfword aligned, `sizeof` is 2

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

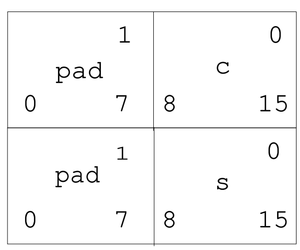

Figure 3-21 `union` Allocation—Little-Endian

Word aligned, `sizeof` is 4

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

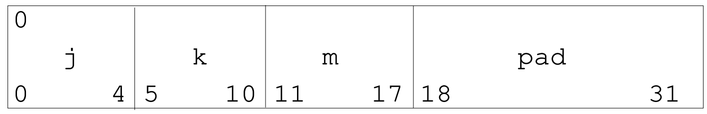

Figure 3-16 Left-to-Right (Big-Endian) Allocation

Word aligned, `sizeof` is 12

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

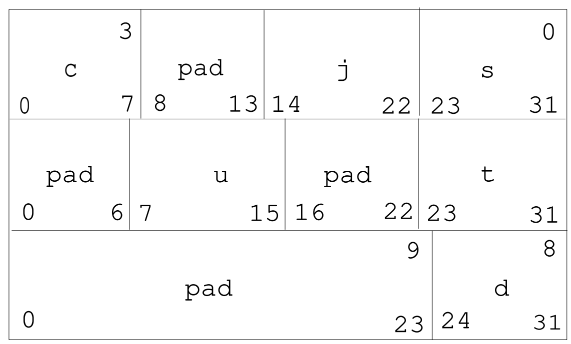

Figure 3-17 Boundary Alignment—Little-Endian

Word aligned, `sizeof` is 12

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

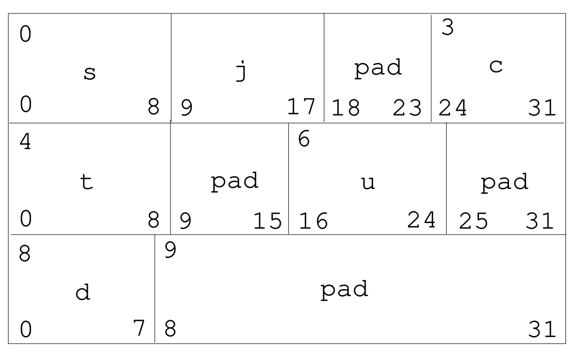

Figure 3-18 Boundary Alignment—Big-Endian

"Plain" bit-fields (that is, those neither `signed` nor `unsigned`) always have non-negative values. Although they may have type `short`, `int`, or `long` (which can have negative values), bit-fields of these types have the same range as bit-fields 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) on Little-Endian implementations and from left to right (most to least significant) on Big-Endian implementations.
- 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 must share a storage unit with other structure and union members (either bit-field or non-bit-field) if and only if there is sufficient space within the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union, although an individual bit-field's member offsets obey the alignment constraints. An unnamed, zero-width bit-field shall prevent any further member, bit-field or other, from residing in the storage unit corresponding to the type of the zero-width bit-field.

The following examples (Figures 3-14 through 3-24) show `struct` and `union` members' byte offsets in the upper right corners for Little-Endian implementations, and in the upper left corners for Big-Endian implementations. Bit numbers appear in the lower corners.

	0 3	1 2	2 1	3 0
0x01020304	01	02	03	04
	0 7	8 15	16 23	24 31

Figure 3-14 Bit Numbering

Word aligned, `sizeof` is 4

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

			0
pad	m	k	j
0 13	14 20	21 26	27 31

Figure 3-15 Right-to-Left (Little-Endian) Allocation

Word aligned, `sizeof` is 4

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

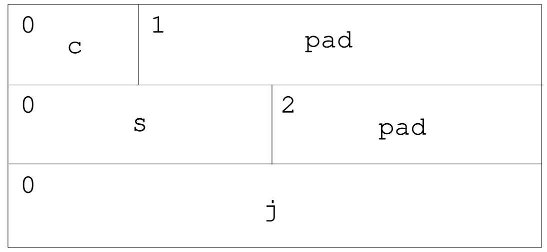

Figure 3-13 `union` Allocation—Big-Endian

Bit-Fields

C `struct` and `union` definitions may have "bit-fields," defining integral objects with a specified number of bits (see Table 3-2).

Table 3-2 Bit-Field Ranges

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

Doubleword aligned, `sizeof` is 24

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

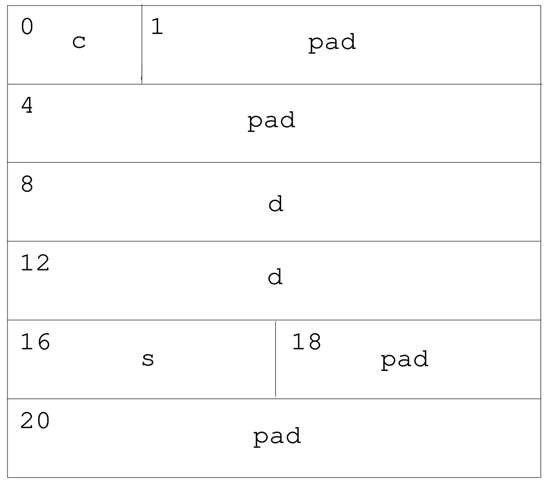

Figure 3-11 Internal and Tail Padding—Big-Endian

Word aligned, `sizeof` is 4

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

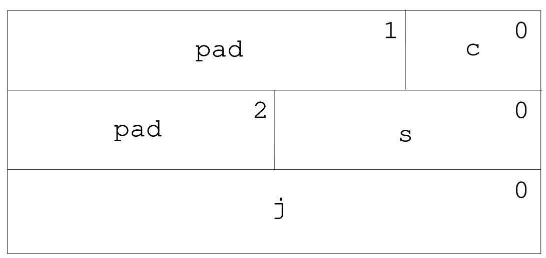

Figure 3-12 `union` Allocation—Little-Endian

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

Halfword aligned, `sizeof` is 4

0	1	2
c	pad	s

Figure 3-9 Internal Padding—Big-Endian

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

Doubleword aligned, `sizeof` is 24

pad 1	c 0
pad 4	
d 8	
d 12	
pad 18	s 16
pad 20	

Figure 3-10 Internal and Tail Padding—Little-Endian

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

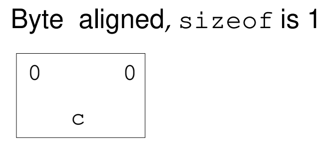

Figure 3-5   Structure Smaller Than a Word

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

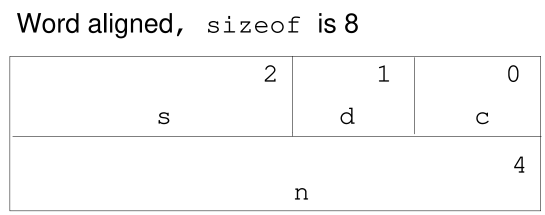

Figure 3-6   No Padding—Little-Endian

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

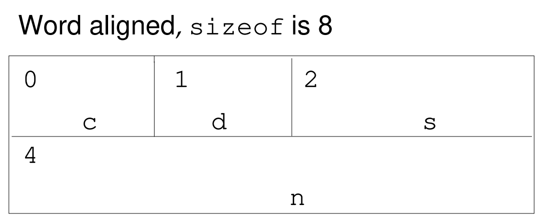

Figure 3-7   No Padding—Big-Endian

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

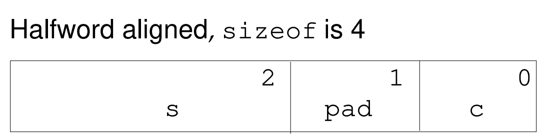

Figure 3-8   Internal Padding—Little-Endian

Table 3-1 Scalar Types *(Continued)*

Type	ANSI C	sizeof	Alignment (bytes)	PowerPC
Floating Point	`float`	4	4	single precision (IEEE)
	`double`	8	8	double precision (IEEE)
	`long double`	16	16	extended precision (IEEE)

Note – "extended precision (IEEE)" in Table 3-1 means IEEE 754 double extended precision with a sign bit, a 15-bit exponent with a bias of -16383, 112 fraction bits (with a leading "implicit" bit).

Note – `long long` and `unsigned long long` data types are implemented by some compilers, although they are not (currently) specified by ANSI C. Programs that use them are not ABI conformant (that is, need not be supported on all platforms that implement this ABI), but if a platform does support the `long long` data types, they shall be implemented as 8-byte quantities aligned on 8-byte boundaries, and treated as specified in the other notes.

Note – Compilers and systems may implement the `long double` data type in some other way for performance reasons, using a compiler option. Examples of such formats could be two successive doubles or even a single double. Such usage does not conform to this ABI, however, and runs the danger of passing a wrongly formatted floating-point number to another, conforming function as an argument. Programs using other formats should transform long double floating-point numbers to a conforming format before putting them in permanent storage.

Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component, that is, the component with the largest alignment. The size of any object, including aggregates and unions, is always a multiple of the alignment of the object. An array uses the same alignment as its elements. Structure and union objects may require padding to meet size and alignment constraints:

- 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.
- If necessary, a structure’s size is increased to make it a multiple of the structure’s alignment. This may require tail padding, depending on the last member.

In the following examples (Figures 3-5 through 3-13), members’ byte offsets for Little-Endian implementations appear in the upper right corners; offsets for Big-Endian implementations in the upper left corners.

0 15 msb 0 7	1 14 8 15	2 13 16 23	3 12 24 31
4 11 32 39	5 10 40 47	6 9 48 55	7 8 56 63
8 7 64 71	9 6 72 79	10 5 80 87	11 4 88 95
12 3 96 103	13 2 104 111	14 1 112 119	15 0 lsb 120 127

Figure 3-4 Bit and Byte Numbering in Quadwords

Fundamental Types

Table 3-1 shows how ANSI C scalar types correspond to those of the PowerPC processor. For all types, a NULL pointer has the value zero.

Table 3-1 Scalar Types

Type	ANSI C	`sizeof`	Alignment (bytes)	PowerPC
Character	`char` `unsigned char`	1	1	unsigned byte
	`signed char`	1	1	signed byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

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

	`unsigned int` `unsigned long`	4	4	unsigned word
Pointer	`any-type *` `any-type (*) ()`	4	4	unsigned word

Figures 3-1 through 3-4 illustrate the conventions for bit and byte numbering within various width storage units. These conventions apply to both integer data and floating-point data, where the most significant byte of a floating-point value holds the sign and at least the start of the exponent. The figures show Little-Endian byte numbers in the upper right corners, Big-Endian byte numbers in the upper left corners, and bit numbers in the lower corners.

Note – In the PowerPC Architecture documentation, the bits in a word are numbered from left to right (MSB to LSB), and figures usually show only the Big-Endian byte order.

---

0 1 msb 0 7	1 0 lsb 8 15

Figure 3-1 Bit and Byte Numbering in Halfwords

0 3 msb 0 7	1 2 8 15	2 1 16 23	3 0 lsb 24 31

Figure 3-2 Bit and Byte Numbering in Words

0 7 msb 0 7	1 6 8 15	2 5 16 23	3 4 24 31
4 3 32 39	5 2 40 47	6 1 48 55	7 0 lsb 56 63

Figure 3-3 Bit and Byte Numbering in Doublewords

3 LOW-LEVEL SYSTEM INFORMATION

Machine Interface

Processor Architecture

The PowerPC Architecture: A Specification for A New Family of RISC Processors defines the PowerPC Architecture. Programs intended to execute directly on the processor use the PowerPC instruction set, and the instruction encodings and semantics of the architecture.

An application program can assume that all instructions defined by the architecture that are neither privileged nor optional exist and work as documented. However, the "Fixed-Point Load and Store Multiple" instructions and the "Fixed-Point Move Assist" instructions are not available in Little-Endian implementations. In Little-Endian mode, the latter groups of instructions always cause alignment exceptions in the PowerPC Architecture; in Big-Endian mode they are usually slower than a sequence of other instructions that have the same effect.

To be ABI-conforming, the processor must implement the instructions of the architecture, perform the specified operations, and produce the expected 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 optional instructions in the PowerPC Architecture, or additional non-PowerPC instructions or capabilities. Programs that use those instructions or capabilities do not conform to the PowerPC ABI; executing them on machines without the additional capabilities gives undefined behavior.

Data Representation

Byte Ordering

The architecture defines an 8-bit byte, a 16-bit halfword, a 32-bit word, a 64-bit doubleword, and a 128-bit quadword. Byte ordering defines how the bytes that make up halfwords, words, doublewords, and quadwords are ordered in memory. Most significant byte (MSB) byte ordering, or "Big-Endian" as it is sometimes called, means that the most significant byte is located in the lowest addressed byte position in a storage unit (byte 0). Least significant byte (LSB) byte ordering, or "Little-Endian" as it is sometimes called, means that the least significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

The PowerPC processor family supports either Big-Endian or Little-Endian byte ordering. This specification defines two ABIs, one for each type of byte ordering. An implementation must state which type of byte ordering it supports.

2 SOFTWARE INSTALLATION

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 media listed below.

- 3.5-inch diskette; 135 TPI (80 tracks/side) double-sided, 18 sectors/track, 512 bytes/sector, total capacity of 1.44 Mbytes per disk
- 3.5-inch diskette; 135 TPI (80 tracks/side) double-sided, 36 sectors/track, 512 bytes/sector, total capacity of 2.88 Mbytes per disk
- 8-mm tape format
- CD-ROM with the ISO 9661 file system format with Rockridge Extensions

a portion of the specification moved to Level 2 support in an edition of the System V 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 both the generic specification and this supplement. All components of the System V ABI and of this supplement have Level 1 support unless they are explicitly labeled as Level 2.

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

- *System V Interface Definition, Issue 3*.
- *The PowerPC Architecture: A Specification for A New Family of RISC Processors.* International Business Machines (IBM). *S*an Francisco: Morgan Kaufmann, 1994.
- *DWARF Debugging Information Format*, Revision: Version 2.0.0, July 27, 1993. UNIX International, Program Languages SIG.

1 INTRODUCTION

The PowerPC Processor and the System V ABI

The System V Application Binary Interface, or System V 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, Issue 3*. This includes systems that have implemented UNIX®System V Release 4.

The System V Application Binary Interface PowerPC™ Processor Supplement (PowerPC Processor ABI Supplement), described in this document, is a supplement to the generic System V ABI, and it contains information specific to System V implementations built on the PowerPC Architecture™ operating in 32-bit mode. The generic System V ABI and this supplement together constitute a complete System V Application Binary Interface specification for systems that implement the 32-bit architecture of the PowerPC processor family.

In the PowerPC Architecture, a processor can run in either of two modes: Big-Endian mode or Little-Endian mode. (See **Byte Ordering** at the beginning of Chapter 3.) Accordingly, this ABI specification really defines two binary interfaces, a Big-Endian ABI and a Little-Endian ABI. Programs and (in general) data produced by programs that run on an implementation of the Big-Endian interface are not portable to an implementation of the Little-Endian interface, and vice versa.

How to Use the PowerPC Processor ABI Supplement

While the generic *System V ABI* is the prime reference document, this document contains PowerPC processor-specific implementation details, some of which supersede information in the generic one.

As with the *System V ABI*, this document refers to other publicly available documents, especially the book titled *IBM PowerPC User Instruction Set Architecture*, all of which should be considered part of this PowerPC Processor ABI Supplement and just as binding as the requirements and data it explicitly includes.

Evolution of the ABI Specification

The System V ABI will evolve over time to address new technology and market requirements, and it will be reissued every three years or so. 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 System V 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. Level 2 support means that a portion of the specification may be withdrawn or altered after the next edition of the System V ABI is made available—that is,

TABLES

FIGURES

CONTENTS

SYSTEM V APPLICATION BINARY INTERFACE
PowerPC Processor Supplement

by
Steve Zucker, SunSoft
Kari Karhi, IBM

September 1995

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 802-3334-10
Revision A, September 1995

SunSoft
A Sun Microsystems, Inc. Business