```
******** include/signal.h ************
/* The <signal.h> header defines all the ANSI and POSIX signals.
 * MINIX supports all the signals required by POSIX. They are defined below.
 * Some additional signals are also supported.
 */

#ifndef _SIGNAL_H
#define _SIGNAL_H

#ifndef _ANSI_H
#include <ansi.h>
#endif
#ifdef _POSIX_SOURCE
#ifndef _TYPES_H
#include <sys/types.h>
#endif
#endif

/* Here are types that are closely associated with signal handling. */
typedef int sig_atomic_t;

#ifdef _POSIX_SOURCE
#ifndef _SIGSET_T
#define _SIGSET_T
typedef unsigned long sigset_t;
#endif
#endif

#define SIGHUP             1    /* hangup */
#define SIGINT             2    /* interrupt (DEL) */
#define SIGQUIT            3    /* quit (ASCII FS) */
#define SIGILL             4    /* illegal instruction */
#define SIGTRAP            5    /* trace trap (not reset when caught) */
#define SIGABRT            6    /* IOT instruction */
#define SIGIOT             6    /* SIGABRT for people who speak PDP−11 */
#define SIGUNUSED          7    /* spare code */
#define SIGFPE             8    /* floating point exception */
#define SIGKILL            9    /* kill (cannot be caught or ignored) */
#define SIGUSR1           10    /* user defined signal # 1 */
#define SIGSEGV           11    /* segmentation violation */
#define SIGUSR2           12    /* user defined signal # 2 */
#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 SIGCHLD           17    /* child process terminated or stopped */

#define SIGEMT             7    /* obsolete */
#define SIGBUS            10    /* obsolete */

/* MINIX specific signals. These signals are not used by user proceses,
 * but meant to inform system processes, like the PM, about system events.
 */
#define SIGKMESS          18    /* new kernel message */
#define SIGKSIG           19    /* kernel signal pending */
#define SIGKSTOP          20    /* kernel shutting down */

/* Regular signals. */
#define SIGWINCH          21    /* window size has changed */

#define _NSIG             21    /* number of signals used */

/* POSIX requires the following signals to be defined, even if they are
 * not supported.  Here are the definitions, but they are not supported.
 */
#define SIGCONT           18    /* continue if stopped */
#define SIGSTOP           19    /* stop signal */
#define SIGTSTP           20    /* interactive stop signal */
#define SIGTTIN           21    /* background process wants to read */
#define SIGTTOU           22    /* background process wants to write */

/* The sighandler_t type is not allowed unless _POSIX_SOURCE is defined. */
typedef void _PROTOTYPE( (*__sighandler_t), (int) );

```

```
/* Macros used as function pointers. */
#define SIG_ERR    ((__sighandler_t) −1)        /* error return */
#define SIG_DFL    ((__sighandler_t)  0)        /* default signal handling */
#define SIG_IGN    ((__sighandler_t)  1)        /* ignore signal */
#define SIG_HOLD   ((__sighandler_t)  2)        /* block signal */
#define SIG_CATCH  ((__sighandler_t)  3)        /* catch signal */
#define SIG_MESS   ((__sighandler_t)  4)        /* pass as message (MINIX) */

#ifdef _POSIX_SOURCE
struct sigaction {
  __sighandler_t sa_handler;    /* SIG_DFL, SIG_IGN, or pointer to function */
  sigset_t sa_mask;             /* signals to be blocked during handler */
  int sa_flags;                 /* special flags */
};

/* Fields for sa_flags. */
#define SA_ONSTACK   0x0001     /* deliver signal on alternate stack */
#define SA_RESETHAND 0x0002     /* reset signal handler when signal caught */
#define SA_NODEFER   0x0004     /* don’t block signal while catching it */
#define SA_RESTART   0x0008     /* automatic system call restart */
#define SA_SIGINFO   0x0010     /* extended signal handling */
#define SA_NOCLDWAIT 0x0020     /* don’t create zombies */
#define SA_NOCLDSTOP 0x0040     /* don’t receive SIGCHLD when child stops */

/* POSIX requires these values for use with sigprocmask(2). */
#define SIG_BLOCK          0    /* for blocking signals */
#define SIG_UNBLOCK        1    /* for unblocking signals */
#define SIG_SETMASK        2    /* for setting the signal mask */
#define SIG_INQUIRE        4    /* for internal use only */
#endif  /* _POSIX_SOURCE */

/* POSIX and ANSI function prototypes. */
_PROTOTYPE( int raise, (int _sig)                                       );
_PROTOTYPE( __sighandler_t signal, (int _sig, __sighandler_t _func)     );

#ifdef _POSIX_SOURCE
_PROTOTYPE( int kill, (pid_t _pid, int _sig)                            );
_PROTOTYPE( int sigaction,
    (int _sig, const struct sigaction *_act, struct sigaction *_oact)   );
_PROTOTYPE( int sigaddset, (sigset_t *_set, int _sig)                   );
_PROTOTYPE( int sigdelset, (sigset_t *_set, int _sig)                   );
_PROTOTYPE( int sigemptyset, (sigset_t *_set)                           );
_PROTOTYPE( int sigfillset, (sigset_t *_set)                            );
_PROTOTYPE( int sigismember, (const sigset_t *_set, int _sig)           );
_PROTOTYPE( int sigpending, (sigset_t *_set)                            );
_PROTOTYPE( int sigprocmask,
            (int _how, const sigset_t *_set, sigset_t *_oset)           );
_PROTOTYPE( int sigsuspend, (const sigset_t *_sigmask)                  );
#endif

#endif /* _SIGNAL_H */

```

```
******** include/sys/sigcontext.h *************
#ifndef _SIGCONTEXT_H
#define _SIGCONTEXT_H

/* The sigcontext structure is used by the sigreturn(2) system call.
 * sigreturn() is seldom called by user programs, but it is used internally
 * by the signal catching mechanism.
 */

#ifndef _ANSI_H
#include <ansi.h>
#endif

#ifndef _MINIX_SYS_CONFIG_H
#include <minix/sys_config.h>
#endif

#if !defined(_MINIX_CHIP)
#include "error, configuration is not known"
#endif

/* The following structure should match the stackframe_s structure used
 * by the kernel's context switching code.  Floating point registers should
 * be added in a different struct.
 */
#if (_MINIX_CHIP == _CHIP_INTEL)
struct sigregs {
#if _WORD_SIZE == 4
  short sr_gs;
  short sr_fs;
#endif /* _WORD_SIZE == 4 */
  short sr_es;
  short sr_ds;
  int sr_di;
  int sr_si;
  int sr_bp;
  int sr_st;                    /* stack top -- used in kernel */
  int sr_bx;
  int sr_dx;
  int sr_cx;
  int sr_retreg;
  int sr_retadr;                /* return address to caller of save -- used
                                 * in kernel */
  int sr_pc;
  int sr_cs;
  int sr_psw;
  int sr_sp;
  int sr_ss;
};

struct sigframe {               /* stack frame created for signalled process */
  _PROTOTYPE( void (*sf_retadr), (void) );
  int sf_signo;
  int sf_code;
  struct sigcontext *sf_scp;
  int sf_fp;
  _PROTOTYPE( void (*sf_retadr2), (void) );
  struct sigcontext *sf_scpcopy;
};

#else
#include "error, _MINIX_CHIP is not supported"
#endif
#endif /* _MINIX_CHIP == _CHIP_INTEL */

struct sigcontext {
  int sc_flags;                 /* sigstack state to restore */
  long sc_mask;                 /* signal mask to restore */
  struct sigregs sc_regs;       /* register set to restore */
};

#if (_MINIX_CHIP == _CHIP_INTEL)
#if _WORD_SIZE == 4
```

```
#define sc_gs sc_regs.sr_gs
#define sc_fs sc_regs.sr_fs
#endif /* _WORD_SIZE == 4 */
#define sc_es sc_regs.sr_es
#define sc_ds sc_regs.sr_ds
#define sc_di sc_regs.sr_di
#define sc_si sc_regs.sr_si
#define sc_fp sc_regs.sr_bp
#define sc_st sc_regs.sr_st             /* stack top -- used in kernel */
#define sc_bx sc_regs.sr_bx
#define sc_dx sc_regs.sr_dx
#define sc_cx sc_regs.sr_cx
#define sc_retreg sc_regs.sr_retreg
#define sc_retadr sc_regs.sr_retadr     /* return address to caller of
                                        save -- used in kernel */
#define sc_pc sc_regs.sr_pc
#define sc_cs sc_regs.sr_cs
#define sc_psw sc_regs.sr_psw
#define sc_sp sc_regs.sr_sp
#define sc_ss sc_regs.sr_ss
#endif /* _MINIX_CHIP == _CHIP_INTEL */

/* Values for sc_flags.  Must agree with <minix/jmp_buf.h>. */
#define SC_SIGCONTEXT   2       /* nonzero when signal context is included */
#define SC_NOREGLOCALS  4       /* nonzero when registers are not to be
                                        saved and restored */

_PROTOTYPE( int sigreturn, (struct sigcontext *_scp)                    );

#endif /* _SIGCONTEXT_H */

```

```
******** lib/posix/_sigaction.c *************
#include <lib.h>
#define sigaction _sigaction
#include <sys/sigcontext.h>
#include <signal.h>

_PROTOTYPE(int __sigreturn, (void));

PUBLIC int sigaction(sig, act, oact)
int sig;
_CONST struct sigaction *act;
struct sigaction *oact;
{
  message m;

  m.m1_i2 = sig;

  /* XXX − yet more type puns because message struct is short of types. */
  m.m1_p1 = (char *) act;
  m.m1_p2 = (char *) oact;
  m.m1_p3 = (char *) __sigreturn;

  return(_syscall(MM, SIGACTION, &m));
}




***************** lib/posix/_sigprocmask.c **************
#include <lib.h>
#define sigprocmask _sigprocmask
#include <signal.h>

PUBLIC int sigprocmask(how, set, oset)
int how;
_CONST sigset_t *set;
sigset_t *oset;
{
  message m;

  if (set == (sigset_t *) NULL) {
        m.m2_i1 = SIG_INQUIRE;
        m.m2_l1 = 0;
  } else {
        m.m2_i1 = how;
        m.m2_l1 = (long) *set;
  }
  if (_syscall(MM, SIGPROCMASK, &m) < 0) return(−1);
  if (oset != (sigset_t *) NULL) *oset = (sigset_t) (m.m2_l1);
  return(m.m_type);
}






*************** lib/posix/_sigsuspend.c *****************
#include <lib.h>
#define sigsuspend _sigsuspend
#include <signal.h>

PUBLIC int sigsuspend(set)
_CONST sigset_t *set;
{
  message m;

  m.m2_l1 = (long) *set;
  return(_syscall(MM, SIGSUSPEND, &m));
}

```

```
*************** lib/posix/_sigreturn.c *************
#include <lib.h>
#define sigfillset      _sigfillset
#define sigjmp          _sigjmp
#define sigprocmask     _sigprocmask
#define sigreturn       _sigreturn
#include <sys/sigcontext.h>
#include <setjmp.h>
#include <signal.h>

_PROTOTYPE( int sigjmp, (jmp_buf jb, int retval));

#if (_SETJMP_SAVES_REGS == 0)
/* ’sigreturn’ using a short format jmp_buf (no registers saved). */
PUBLIC int sigjmp(jb, retval)
jmp_buf jb;
int retval;
{
  struct sigcontext sc;

  sc.sc_flags = jb[0].__flags;
  sc.sc_mask = jb[0].__mask;

#if (CHIP == INTEL)
  sc.sc_pc = (int) jb[0].__pc;
  sc.sc_sp = (int) jb[0].__sp;
  sc.sc_fp = (int) jb[0].__lb;
#endif

  sc.sc_retreg = retval;
  return sigreturn(&sc);
}
#endif

PUBLIC int sigreturn(scp)
register struct sigcontext *scp;
{
  sigset_t set;

  /* The message can’t be on the stack, because the stack will vanish out
   * from under us.  The send part of sendrec will succeed, but when
   * a message is sent to restart the current process, who knows what will
   * be in the place formerly occupied by the message?
   */
  static message m;

  /* Protect against race conditions by blocking all interrupts. */
  sigfillset(&set);             /* splhi */
  sigprocmask(SIG_SETMASK, &set, (sigset_t *) NULL);

  m.m2_l1 = scp->sc_mask;
  m.m2_i2 = scp->sc_flags;
  m.m2_p1 = (char *) scp;
  return(_syscall(MM, SIGRETURN, &m));  /* normally this doesn’t return */
}




****************** lib/i386/rts/__sigreturn.s ******
! This routine is the low−level code for returning from signals.
! It calls __sigreturn, which is the normal "system call" routine.
! Both ___sigreturn and __sigreturn are needed.
.sect .text; .sect .rom; .sect .data; .sect .bss
.sect .text
.define ___sigreturn
.extern __sigreturn
___sigreturn:
        add esp, 16
        jmp __sigreturn

```

```
****************** lib/posix/_sigset.c **************
#include <lib.h>
/* XXX − these have to be hidden because signal() uses them and signal() is
 * ANSI and not POSIX.  It would be surely be better to use macros for the
 * library and system uses, and perhaps macros as well as functions for the
 * POSIX user interface.  The macros would not need underlines.  It may be
 * inconvenient to match the exact semantics of the current functions
 * because the interface is bloated by reporting errors.  For library and
 * system uses, the signal number is mostly already known to be valid
 * before the sigset−changing routines are called.
 */
#define sigaddset       _sigaddset
#define sigdelset       _sigdelset
#define sigemptyset     _sigemptyset
#define sigfillset      _sigfillset
#define sigismember     _sigismember
#include <signal.h>

/* Low bit of signal masks. */
#define SIGBIT_0        ((sigset_t) 1)

/* Mask of valid signals (0 − _NSIG). */
#define SIGMASK         (((SIGBIT_0 << _NSIG) << 1) − 1)

#define sigisvalid(signo) ((unsigned) (signo) <= _NSIG)

PUBLIC int sigaddset(set, signo)
sigset_t *set;
int signo;
{
  if (!sigisvalid(signo)) {
        errno = EINVAL;
        return −1;
  }
  *set |= SIGBIT_0 << signo;
  return 0;
}

PUBLIC int sigdelset(set, signo)
sigset_t *set;
int signo;
{
  if (!sigisvalid(signo)) {
        errno = EINVAL;
        return −1;
  }
  *set &= ~(SIGBIT_0 << signo);
  return 0;
}

PUBLIC int sigemptyset(set)
sigset_t *set;
{
  *set = 0;
  return 0;
}

PUBLIC int sigfillset(set)
sigset_t *set;
{
  *set = SIGMASK;
  return 0;
}

PUBLIC int sigismember(set, signo)
_CONST sigset_t *set;
int signo;
{
  if (!sigisvalid(signo)) {
        errno = EINVAL;
        return −1;
  }
  if (*set & (SIGBIT_0 << signo))
```

```
        return 1;
  return 0;
}






********************** lib/posix/_sigpending.c *******************
#include <lib.h>
#define sigpending _sigpending
#include <signal.h>

PUBLIC int sigpending(set)
sigset_t *set;
{
  message m;

  if (_syscall(MM, SIGPENDING, &m) < 0) return(−1);
  *set = (sigset_t) m.m2_l1;
  return(m.m_type);
}
```

```
****************** kernel/system/do_sigsend.c **************
/* The kernel call that is implemented in this file:
 *   m_type:    SYS_SIGSEND
 *
 * The parameters for this kernel call are:
 *     m2_i1:   SIG_PROC        # process to call signal handler
 *     m2_p1:   SIG_CTXT_PTR    # pointer to sigcontext structure
 *     m2_i3:   SIG_FLAGS       # flags for S_SIGRETURN call
 *
 */

#include "../system.h"
#include <signal.h>
#include <string.h>
#include <sys/sigcontext.h>

#if USE_SIGSEND

/*===========================================================================*
 *                              do_sigsend                                   *
 *===========================================================================*/
PUBLIC int do_sigsend(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_sigsend, POSIX−style signal handling. */

  struct sigmsg smsg;
  register struct proc *rp;
  phys_bytes src_phys, dst_phys;
  struct sigcontext sc, *scp;
  struct sigframe fr, *frp;

  if (! isokprocn(m_ptr−>SIG_PROC)) return(EINVAL);
  if (iskerneln(m_ptr−>SIG_PROC)) return(EPERM);
  rp = proc_addr(m_ptr−>SIG_PROC);

  /* Get the sigmsg structure into our address space.  */
  src_phys = umap_local(proc_addr(PM_PROC_NR), D, (vir_bytes)
      m_ptr−>SIG_CTXT_PTR, (vir_bytes) sizeof(struct sigmsg));
  if (src_phys == 0) return(EFAULT);
  phys_copy(src_phys,vir2phys(&smsg),(phys_bytes) sizeof(struct sigmsg));

  /* Compute the user stack pointer where sigcontext will be stored. */
  scp = (struct sigcontext *) smsg.sm_stkptr − 1;

  /* Copy the registers to the sigcontext structure. */
  memcpy(&sc.sc_regs, (char *) &rp−>p_reg, sizeof(struct sigregs));

  /* Finish the sigcontext initialization. */
  sc.sc_flags = SC_SIGCONTEXT;
  sc.sc_mask = smsg.sm_mask;

  /* Copy the sigcontext structure to the user’s stack. */
  dst_phys = umap_local(rp, D, (vir_bytes) scp,
      (vir_bytes) sizeof(struct sigcontext));
  if (dst_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));

  /* Initialize the sigframe structure. */
  frp = (struct sigframe *) scp − 1;
  fr.sf_scpcopy = scp;
  fr.sf_retadr2= (void (*)()) rp−>p_reg.pc;
  fr.sf_fp = rp−>p_reg.fp;
  rp−>p_reg.fp = (reg_t) &frp−>sf_fp;
  fr.sf_scp = scp;
  fr.sf_code = 0;       /* XXX − should be used for type of FP exception */
  fr.sf_signo = smsg.sm_signo;
  fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;

  /* Copy the sigframe structure to the user’s stack. */
  dst_phys = umap_local(rp, D, (vir_bytes) frp,
      (vir_bytes) sizeof(struct sigframe));
  if (dst_phys == 0) return(EFAULT);
```

```
  phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));

  /* Reset user registers to execute the signal handler. */
  rp−>p_reg.sp = (reg_t) frp;
  rp−>p_reg.pc = (reg_t) smsg.sm_sighandler;

  return(OK);
}

#endif /* USE_SIGSEND */

```

```
**************** kernel/system/do_sigreturn.c ***************
/* The kernel call that is implemented in this file:
 *   m_type:    SYS_SIGRETURN
 *
 * The parameters for this kernel call are:
 *     m2_i1:   SIG_PROC        # process returning from handler
 *     m2_p1:   SIG_CTXT_PTR    # pointer to sigcontext structure
 *
 */

#include "../system.h"
#include <string.h>
#include <signal.h>
#include <sys/sigcontext.h>

#if USE_SIGRETURN

/*===========================================================================*
 *                              do_sigreturn                                 *
 *===========================================================================*/
PUBLIC int do_sigreturn(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* POSIX style signals require sys_sigreturn to put things in order before
 * the signalled process can resume execution
 */
  struct sigcontext sc;
  register struct proc *rp;
  phys_bytes src_phys;

  if (! isokprocn(m_ptr->SIG_PROC)) return(EINVAL);
  if (iskerneln(m_ptr->SIG_PROC)) return(EPERM);
  rp = proc_addr(m_ptr->SIG_PROC);

  /* Copy in the sigcontext structure. */
  src_phys = umap_local(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
      (vir_bytes) sizeof(struct sigcontext));
  if (src_phys == 0) return(EFAULT);
  phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));

  /* Make sure that this is not just a jump buffer. */
  if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);

  /* Fix up only certain key registers if the compiler doesn’t use
   * register variables within functions containing setjmp.
   */
  if (sc.sc_flags & SC_NOREGLOCALS) {
      rp->p_reg.retreg = sc.sc_retreg;
      rp->p_reg.fp = sc.sc_fp;
      rp->p_reg.pc = sc.sc_pc;
      rp->p_reg.sp = sc.sc_sp;
      return(OK);
  }
  sc.sc_psw  = rp->p_reg.psw;

#if (CHIP == INTEL)
  /* Don’t panic kernel if user gave bad selectors. */
  sc.sc_cs = rp->p_reg.cs;
  sc.sc_ds = rp->p_reg.ds;
  sc.sc_es = rp->p_reg.es;
#if _WORD_SIZE == 4
  sc.sc_fs = rp->p_reg.fs;
  sc.sc_gs = rp->p_reg.gs;
#endif
#endif

  /* Restore the registers. */
  memcpy(&rp->p_reg, &sc.sc_regs, sizeof(struct sigregs));
  return(OK);
}
#endif /* USE_SIGRETURN */

```

```
============== kernel/system/do_kill.c ============
/* The kernel call that is implemented in this file:
 *   m_type:    SYS_KILL
 *
 * The parameters for this kernel call are:
 *     m2_i1:   SIG_PROC        # process to signal/ pending
 *     m2_i2:   SIG_NUMBER      # signal number to send to process
 */

#include "../system.h"
#include <signal.h>
#include <sys/sigcontext.h>

#if USE_KILL

/*===========================================================================*
 *                                do_kill                                    *
 *===========================================================================*/
PUBLIC int do_kill(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_kill(). Cause a signal to be sent to a process. The PM is the
 * central server where all signals are processed and handler policies can
 * be registered. Any request, except for PM requests, is added to the map
 * of pending signals and the PM is informed about the new signal.
 * Since system servers cannot use normal POSIX signal handlers (because they
 * are usually blocked on a RECEIVE), they can request the PM to transform
 * signals into messages. This is done by the PM with a call to sys_kill().
 */
  proc_nr_t proc_nr = m_ptr->SIG_PROC;
  int sig_nr = m_ptr->SIG_NUMBER;

  if (! isokprocn(proc_nr) || sig_nr > _NSIG) return(EINVAL);
  if (iskerneln(proc_nr)) return(EPERM);

  if (m_ptr->m_source == PM_PROC_NR) {
      /* Directly send signal notification to a system process. */
      if (! (priv(proc_addr(proc_nr))->s_flags & SYS_PROC)) return(EPERM);
      send_sig(proc_nr, sig_nr);
  } else {
      /* Set pending signal to be processed by the PM. */
      cause_sig(proc_nr, sig_nr);
  }
  return(OK);
}

#endif /* USE_KILL */

```

```
=========== kernel/system/do_getksig.c ===========
/* The kernel call that is implemented in this file:
 *   m_type:    SYS_GETKSIG
 *
 * The parameters for this kernel call are:
 *     m2_i1:   SIG_PROC        # process with pending signals
 *     m2_l1:   SIG_MAP         # bit map with pending signals
 */

#include "../system.h"
#include <signal.h>
#include <sys/sigcontext.h>

#if USE_GETKSIG

/*===========================================================================*
 *                              do_getksig                                   *
 *===========================================================================*/
PUBLIC int do_getksig(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* PM is ready to accept signals and repeatedly does a kernel call to get
 * one. Find a process with pending signals. If no signals are available,
 * return NONE in the process number field.
 * It is not sufficient to ready the process when PM is informed, because
 * PM can block waiting for FS to do a core dump.
 */
  register struct proc *rp;

  /* Find the next process with pending signals. */
  for (rp = BEG_USER_ADDR; rp < END_PROC_ADDR; rp++) {
      if (rp->p_rts_flags & SIGNALED) {
          m_ptr->SIG_PROC = rp->p_nr;           /* store signaled process */
          m_ptr->SIG_MAP = rp->p_pending;       /* pending signals map */
          sigemptyset(&rp->p_pending);          /* ball is in PM’s court */
          rp->p_rts_flags &= ~SIGNALED;         /* blocked by SIG_PENDING */
          return(OK);
      }
  }

  /* No process with pending signals was found. */
  m_ptr->SIG_PROC = NONE;
  return(OK);
}
#endif /* USE_GETKSIG */

```

```
============= kernel/system/do_endksig.c =============
/* The kernel call that is implemented in this file:
 *   m_type:    SYS_ENDKSIG
 *
 * The parameters for this kernel call are:
 *     m2_i1:   SIG_PROC        # process for which PM is done
 */

#include "../system.h"
#include <signal.h>
#include <sys/sigcontext.h>

#if USE_ENDKSIG

/*===========================================================================*
 *                              do_endksig                                   *
 *===========================================================================*/
PUBLIC int do_endksig(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Finish up after a kernel type signal, caused by a SYS_KILL message or a
 * call to cause_sig by a task. This is called by the PM after processing a
 * signal it got with SYS_GETKSIG.
 */
  register struct proc *rp;

  /* Get process pointer and verify that it had signals pending. If the
   * process is already dead its flags will be reset.
   */
  rp = proc_addr(m_ptr->SIG_PROC);
  if (! (rp->p_rts_flags & SIG_PENDING)) return(EINVAL);

  /* PM has finished one kernel signal. Perhaps process is ready now? */
  if (! (rp->p_rts_flags & SIGNALED))           /* new signal arrived */
     if ((rp->p_rts_flags &= ~SIG_PENDING)==0)  /* remove pending flag */
         lock_enqueue(rp);                      /* ready if no flags */
  return(OK);
}

#endif /* USE_ENDKSIG */


```