when any key is pressed only if the `modem_f()` is not currently running.

The guard in Slot 4 is set to ignore Traps 32, 33 and 57. If the Slot 4 function, `modem_f()`, is running when a key press occurs (either Trap 32 or 33), the `keys_f()` is pended. Once the `modem_f()` completes, the system immediately calls and executes `keys_f()` before returning to the original interrupt point.

Since traps are not buffered, only one event can be pended per trap number. For example, if during the execution of `modem_f()` the [CLEAR] key is pressed three times, only one [CLEAR] key event will be pending. However, three clear keys (0X1B) are placed in the keyboard buffer. This situation is not uncommon, and stresses why the programmer must be aware that a one-to-one correlation does not always exist between guard function calls and bytes in the device buffers.

If multiple traps are pended, they are processed from the lowest to highest trap number and not in the order in which they occurred. Thus, if a [CLEAR] key event (Trap 32) and a card swipe event (Trap 57) are both pending, the [CLEAR] key event is processed first, regardless of the order in which the events occurred.

Events and their associated guard functions are only pended when another guard function is running that has some or all events listed as traps to ignore. During normal application execution, events are immediately processed and are never pended.

❖ *Since pending events are often difficult to manage, keep guard functions small and quick. Waits and delays should never be part of a trap handling routine.*

Trap Mask Functions

The TXO library includes several functions to assist in the creation and manipulation of the exception handling system's bit-mapped trap masks. Trap masks are defined in the <trap.h> header file in the structure trap_mask:

```
struct trap_mask {unsigned char a[8];};
```

Examples of each of the TXO trap mask functions are included in *Chapter 11, Library Functions*. The following is a list of these functions.

`mask_empty()..` Returns a mask with all mask positions set to 0.

`mask_fill()...` Returns a mask with all mask positions set to 1.

`mask_of().....` Creates a mask with a single bit on, it corresponds to an integer which must be in the range 0 to 63.

`mask_and()....` Returns a mask formed by ANDing two masks.

`mask_or().....` Returns a mask formed by ORing two masks.

`mask_clear()..` Returns a mask formed by taking one mask and turning OFF all bits which are ON in a second mask.

`mask_in().....` Tests a specified bit in a mask. It returns true (1) if the bit is set and false (0) if it is off.

Creating Guard Functions

The event/exception-handling system maintains an internal bit map of 64 bits called the system mask. Each bit is associated with one of the 64 events, or traps, defined in the system. When an event or exception occurs, its associated bit in the system mask is set. To determine how the event will be handled, the system mask is ANDed with the each guard's trap mask. The first TRUE value returned will cause the associated guard function to be called.

The integer value of the active trap is passed to the guard function being called. For example, when a card swipe (Trap 57) causes a guard function to be called, the value 57 is passed to that guard function. Since a single guard can monitor multiple traps, the value is used by the guard function to determine which event caused its execution.

As shown in the example on the following page, all functions called from the guard list should be of the type `void()` as there is no valid way to return a value from these functions.

When an event being monitored by a guard occurs, the associated guard function is immediately called. Current processing is suspended and the associated guard function is added to the stack. Processing will not return to the original point until the guard function completes.

If during the execution of a guard function another event occurs, it is possible that a second copy of the same guard function (or a new one) will begin running. If this happens, the currently running function will be suspended and the new guard function is added to the stack. Recursion is only limited by the amount of stack space.

To better manage this functionality, the system provides a mechanism to allow events to be pended until the currently running guard function completes. See "Pending Events" earlier in this section, noting its caveat that, due to the nature of pended events, guard functions should be small and quick.

```
/* Example Guard Function */
void many_traps(trap_num)
 int trap_num;
   {
     if(ignore_trap!=1)
     {
       switch(trap_num) {
         case 57: /* card swipe   */
              .
              .
            break;
         case 33: /* key press    */
              .
              .
            break;
         case 40: /* carrier loss */
              .
              .
            break;
         }
       }
     }
```

Tips for Writing Guard Functions

Keep it Simple

Keep functions small and quick. Set flags and read buffers only. Avoid prompting for data input.

It is preferable to decrease the possibility of pending events. The best way to accomplish this to ensure guard functions are very small and quick. Set a flag to signal the event occurred and, if possible, read the data into a temporary buffer from the operating system buffer. All other processing should be handled in the body of the code based on the conditional flag which was set in the guard function. See the section on [CLEAR] key trapping.

Arming Guards

Be careful not to accidentally loop an `arm_guard()` call. The system allows 16 guards to be armed at any one time.

By accidentally looping an `arm_guard()` call, two, three or more of the same guard could be needlessly armed in the system. Since only one guard function is called per event, this will not cause an error until all 16 slots are used. When 17 guards are armed, the `arm_guard()` call will return -1.

Disarming Guards

Remember that disarming and re-arming a guard can change the seniority of the guard list. In the following example, assume guard one is disarmed and then later re-armed—making it the youngest guard in the guard list and, therefore, the first guard to be examined when an event occurs. If the [CLEAR] key is pressed the function `clear_key()` will be called. The `keys_f()` will only be called if other keys (Trap 33) are pressed.

Example Guard List

Slot Number	Guard Function	Traps to Watch	Traps to Ignore
4	`modem_f()`	41, 49	32, 33, 57
3 (oldest)	`keys_f()`	33, 32	All
2	`card_f()`	57	None
1 (youngest)	`clear_key()`	32	None

There are three ways to temporarily "disarm" a guard without using the library function `disarm_guard()`:

- Arm a temporary guard to intercept the event. Any guard added to the guard list will be the youngest guard. A temporary guard can be used to mask an event from triggering an older guard. To reinstate the older guard, simply disarm the temporary guard.
- Set an IGNORE flag in the guard function. As shown earlier, a flag can be set to cause the guard function to immediately return without processing.
- Ignore any flags set by the guard functions. The best way to write a guard function is to have it simply set flags and read data into temporary buffers. Since the body of the application must check these flags before executing conditional processing, the application can simply ignore these flags when needed.

Avoid `Setjmp()` *and* `Longjmp()`

Avoid using `setjmp()` and `longjmp()` calls, as they can have undesirable effects on RAM usage. For example, if a generic library function that mallocs memory is running, and an event which invokes a guard function containing a `longjmp()` occurs before the malloc'ed memory is freed, the malloc'ed memory will never be freed by the system. A similar problem can occur with file and device handles, as well.

To avoid this situation, the application should check for event flags which are set by the guard functions and then cascade via `returns()` to higher levels. See [CLEAR] key processing later in this section.

Use Device Traps Sparingly

It is not necessary to trap every input. Use traps in combination with `while()` loops to poll for device input. When using device traps keep in mind the following:

- Writing to the keyboard or card reader will cause a trap to occur on those devices.
- Devices other than `STDIN`, `STDOUT`, and `STDERR` must be opened for traps to occur.
- If a card swipe is pending in the card reader, the card reader device is temporarily disabled and subsequent swipes will not cause a trap to occur. Once the data has been read from the card reader, subsequent swipes will cause a trap.
- When using the modem (or other communication device) in packet mode, each packet is considered one event. Each byte received that is not part of a packet is treated as an event.

Clear Key Processing

In most POS applications, the [CLEAR] key is used to signal an abort of the current transaction. To properly implement use of the [CLEAR] key, you need to arm a guard at the appropriate point in `main()` to monitor the [CLEAR] key presses (Trap 32) using code similar to the following:

```
#include <trap.h>
int slot;

slot=arm_guard(trap_clr_key,mask_of(32),mask_fill());
```

The `mask_fill()` argument in the third parameter ensures that the guard function `trap_clr_key()` will not be interrupted by any other guard function.

The function `trap_clr_key()` looks like this:

```
void trap_clr_key()
 {
 char temp_buff[120];
 gu_clr_state=1;   /* Clear key flag checked by application */
 read(STDIN,temp_buff,20);  /* Flush keyboard buffer */
 read(card_h,temp_buff,120);/* Flush card reader buffer */
   .
   . /* Flush other OS buffers */
   .
 return();
 }
```

The `trap_clr_key()` call assumes that `gu_clr_state` and `card_h` are global variables, and that `card_h` is the handle to the opened card reader device.

All other abort processing should be internal to each function in the main body of the application as illustrated in the following code:

```
int input_exp_date()
 {
 exp_buff[5];
 int ret_val=0;/* Return values to next level
        -1 Device error
        -2 Invalid exp date
        -3 Clear key pressed during function
         1 Exp date valid */
 .
 .
 .
 clrscr();
 write(STDOUT,"ENTER EXP DATE",14);

 /* Wait for one key, but leave key in OS buffer */
 while(!ioctl(STDIN,GetCtrl|0,0) && !gu_clr_state);
 if(!gu_clr_state)
   {
   /* get expiration date */
   if(-1==SVC_KEY_NUM(exp_buff,0,4,4))
     ret_val=-3;
   else ret_val=validate_exp(exp_buff);  /* demo function call */
   }
   else
   {
   /* do all internal function housekeeping here */
   ret_val=-3;
   }
 .
 .
 .
 return(ret_val);
}
```

The preceding code provides three methods to check for a [CLEAR] key press:

- Check the global flag `gu_clr_state`, which is set in `trap_clr_key()`.
- Check the return value from the library routines `SVC_KEY_TXT()` or `SVC_KEY_NUM()`. Note that when

these functions are called the [CLEAR] key trap is disabled.

- Check the return value from other functions.

Also note that the housekeeping for this function is placed internal to the function and not in the `trap_clr_key()` function. The benefits of this approach include:

- Keeping `trap_clr_key()` generic, quick and easier to test.
- Making this function reusable in other applications.
- Simplifying code maintenance—it is easier to maintain code if it is all in one place.

Trap Functions

The following functions are used to implement the exception-handling system. Refer to Section 10, *Library Functions*, for more details on each function.

arm_guard()

"Activates" a guard by specifying the events it will monitor and identifying the guard function to be called should the event occur.

disarm_guard()

Achieves the reverse of `arm_guard()`.

pending_traps()

Copies the system mask (to identify pending events) and resets any or all pending events to avoid guard activation.

After copying the system mask to an application mask, use the `mask_in()` library function to determine which events are pending in the system. The System Mask is the internal mask where pending events are set.

dispatch_guard()

Calls the guard function defined in the `arm_guard()` routine. See *Using dispatch_guard* below.

which_guard()

Returns the guard function reference for the most recently enabled guard associated with a particular trap.

See *Using which_guard()* on the following page.

Using dispatch_guard()

User Traps 28-31 are set by calling the function `dispatch_guard()`. For example, the call:

```
dispatch_guard(mask_of(28));
```

will force the activation of the youngest guard that has been armed to monitor for Trap 28. This ability gives generic code modules a way to invoke functions without knowing the function's name. This ability is useful when more than one person is working on an application.

For example, assume two people are working on an application. Sally is responsible for the bulk of the code while Bob is writing support functions. Bob needs to call error handling functions should any device errors occur in his code. Sally plans to have the error handling functions change, depending on where in the application the error occurs. She tells Bob to call:

```
dispatch_guard(mask_of(28));
```

in his code whenever a device error occurs. Throughout the main body of the code Sally can now disarm and re-arm guards tying different functions to Trap 28. Thus, the error handling function becomes contextual to the transaction processing. Bob need only call the `dispatch_guard()` function to invoke the correct error handling process.

Using which_guard()

Suppose we have two uses for the [CLEAR] key: 1) abort transaction; or 2) clear display during data entry. We might then have two different guard procedures. The first would be the application-wide "abort transaction" routine which uses `longjmp()`. The second would be a routine installed, by `arm_guard()`, only while the application is in data entry mode. If there is a need to distinguish which of these two routines will receive this trap, `which_guard()` is used.

The application designer, understanding the normal sequence of processing, would be aware of which guard was active, and would know how to take care of this situation. However, a generic library routine would not know about such a sequence and would use `which_guard()`.

Exception Handling Examples

The following pages include three example programs showing exception handling implementations.

Exception Handling Example #1, GUARDS.C

```
/* GUARDS.C */

#include <stdio.h>
#include <trap.h>
#include <txosvc.h>

/* THIS PROGRAM DEMONSTRATES TRAPS AND GUARDS.  THE FIRST
   FUNCTION clear_key() IS CALLED WHEN THE SYSTEM TRAPS
   A CLEAR KEY PRESS FROM THE KEYPAD.  ALL TRAPS ARE IGNORED
   DURING THE EXECUTION OF clear_key().  BEFORE RETURNING,
   clear_key() CHECKS TO SEE IF ANY TRAPS OCCURRED DURING ITS
   EXECUTION AND RESETS ALL TRAPS.  IF A CLEAR KEY WAS PRESSED
   IT WILL PRINT "CLEAR RESET".  IF ANY OTHER KEY WAS PRESSED
   IT WILL CALL THE FUNCTION other_key().

   THE FUNCTION other_key() SIMPLY PRINTS "OTHER KEY".  ALL TRAPS
   ARE MONITORED DURING THIS FUNCTION.  IF A CLEAR KEY PRESS OCCURS
   DURING THIS FUNCTION, THE clear_key() FUNCTION WILL BE CALLED.
   IF A KEY PRESS OTHER THAN THE CLEAR KEY OCCURS THE FUNCTION
   other_key() WILL BE EXECUTED IMMEDIATELY.  BE AWARE THAT SINCE ALL
   TRAPS ARE MONITORED DURING THE EXECUTION OF other_key() THE
   FUNCTION CAN BE CALLED FROM ITSELF REPEATEDLY.  THIS RECURSION
   COULD BLOW THE STACK.  */

/********************** CLEAR KEY FUNCTION ********************/

void clear_key ()
      {
        struct trap_mask sys_mask,reset_mask;
        int i;
        printf ("\fCLEAR KEY ");

/* WAIT FOR KEY PRESSES BY PRINTING FIVE PERIODS.  ANY KEYS PRESSED
   DURING THIS WAITING TIME ARE PENDED BY THE SYSTEM  */
```

```
        for (i=0;i<6;++i)
            { printf (".");
              SVC_WAIT (500); }
        reset_mask = mask_fill();

/* GET ALL PENDING TRAPS AND RESET ALL TRAPS */
        pending_traps(&sys_mask,&reset_mask);

/* IF TRAP 32 (CLEAR) WAS SET PRINT "CLEAR RESET" IF ANY OTHER
   KEY WAS PRESSED PRINT "CALLING..." AND EXECUTE other_key() */
        if (mask_in(sys_mask,32)) printf ("\fCLEAR RESET ");
        if (mask_in(sys_mask,33))
          { printf("\fCALLING ... ");
            SVC_WAIT (1000);

/* DISPATCH GUARD WILL EXECUTE other_key() BECAUSE other_key()
   IS THE FUNCTION ASSOCIATED WITH THE MOST RECENTLY SET GUARD
   THAT MONITORS TRAP 33 */
            dispatch_guard(mask_of(33)); }

      }
/********************** OTHER KEY FUNCTIONS ******************/

void other_key ()
     {
        printf ("\fOTHER KEY ");
        SVC_WAIT (1000);

     }

/****************************** PROC ***********************/

void proc ()
     {
        printf ("\fYOU'RE HERE ");
        SVC_WAIT (1000);
     }


/************************ MAIN ****************************/

main ()
     {
```

```
      struct trap_mask mask1,mask2,mask3,
                       sys_mask, hide_all,
                       hide_none;
      int slot2, slot3, slot4,i;
      printf ("\f");  /* A FORM FEED CLEARS A ONE LINE DISPLAY */
      mask1 = mask_of(32);    /* SET BIT 32 (CLEAR KEY)  */
      mask2 = mask_of(33);    /* SET BIT 33 (OTHER KEYS) */

/* ORing MASK1 AND MASK2 CREATES A MASK3 WITH BOTH BITS 32 AND
   33 SET (CLEAR & OTHER KEY) */
      mask3 = mask_or(mask1,mask2);

/* SET hide_all TO A SET OF 64 ON BITS */
      hide_all = mask_fill();

/* SET hide_none TO A SET OF 64 OFF BITS */
      hide_none = mask_empty();

/* ARM A GUARD TO EXECUTE THE FUNCTION proc() IF EITHER A CLEAR KEY
   OR ANY OTHER KEY IS PRESSED AND IGNORE ALL TRAPS DURING
   EXECUTION */
      slot2 = arm_guard (proc,mask3,hide_all);
      if (-1 == slot2) printf ("\fERROR");

/* ARM A GUARD TO EXECUTE THE FUNCTION clear_key() IF THE CLEAR
   KEY IS PRESSED AND IGNORE ALL TRAPS DURING EXECUTION */
      slot3 = arm_guard (clear_key,mask1,hide_all);
      if (-1 == slot3) printf ("\fERROR");

/* ARM A GUARD TO EXECUTE THE FUNCTION other_key() IF ANY KEY
   OTHER THAN THE CLEAR KEY IS PRESSED BUT DO NOT IGNORE OTHER
   TRAPS DURING EXECUTION */
      slot4 = arm_guard (other_key,mask2,hide_none);
      if (-1 == slot4) printf ("\fERROR");

/* NOTE: THE FUNCTION proc() WILL NEVER BE CALLED BY ITS ASSOCIATED
         GUARD UNTIL ONE OF THE OTHER TWO GUARDS IS DISARMED.  THIS
         IS BECAUSE THE GUARDS ARE EXAMINED FROM MOST RECENT TO
         OLDEST. ON ANY KEY PRESS EITHER clear_key() or
other_key()
         WILL BE EXECUTED BEFORE THE GUARD IN SLOT 3 IS EXAMINED BY
         THE SYSTEM. */

/* PRINT NUMBER WHILE USER CAN TEST TRAPPING */
```

```
    for (i=0;i<100;++i)
        {
          printf("%d ",i);
          SVC_WAIT (500);

/* WHEN I = 50 DISARM GUARD IS SLOT 3 WHICH CALLS clear_key().
   NOW A CLEAR KEY PRESS WILL CALL proc() */
          if (50 == i) disarm_guard(slot3);
        }
     printf ("\fTHE END");
    }
```

Exception Handling Example #2, TRAPPER.C

```
/* TRAPPER.C */

/*********************************************************/
/* This example arms two trap handlers: one to handle    */
/* exceptions and the other to handle normal happenings. */
/* The exception handler takes care of the CLEAR key. The */
/* normal handler takes care of the 1 second timer and   */
/* handles IO from the keyboard (not CLEAR), magcard,    */
/* and barcode readers. The CLEAR key processing shows   */
/* a typical usage of setjmp() and longjmp().            */
/*********************************************************/

/* The following interpreter trap definitions are defined
   in <TRAP.H> in newer releases of TXO Workbench.       */

#define TRAP_STACK_OVERFLOW      16
#define TRAP_HEAP_ERROR          17
#define TRAP_ILLEGAL_INS         18
#define TRAP_BAD_PC              23
```

```
/* Four user defined traps. */
#define TRAP_USER4                28
#define TRAP_USER3                29
#define TRAP_USER2                30
#define TRAP_USER1                31

/* Traps 32..63 are reserved for devices. */
#define TRAP_KEY_CLEAR            32
#define TRAP_KEY_OTHER            33
#define TRAP_CLOCK_SEC            38
#define TRAP_CLOCK_TICK           39
#define TRAP_MODEM_NO_CARRIER     40
#define TRAP_MODEM_INPUT_AVAIL    41
#define TRAP_COM1_OUTPUT_FAIL     42
#define TRAP_COM1_INPUT_AVAIL     43
#define TRAP_COM2_OUTPUT_FAIL     44 /* Not avail on 3XX terminals */
#define TRAP_COM2_INPUT_AVAIL     45 /* Not avail on 3XX terminals */
#define TRAP_COM3_OUTPUT_FAIL     46
#define TRAP_COM3_INPUT_AVAIL     47
#define TRAP_COM4_OUTPUT_FAIL     48
#define TRAP_COM4_INPUT_AVAIL     49
#define TRAP_LAN_EXCEPT_EVENT     52
#define TRAP_LAN_INPUT_AVAIL      53
#define TRAP_MAG_CARD_AVAIL       57
#define TRAP_BAR_CODE_AVAIL       59
#define TRAP_PINPAD_OUTPUT_FAIL   60
#define TRAP_PINPAD_INPUT_AVAIL   61

#include <stdio.h>
#include <device.h>
#include <ascii.h>
#include <config.h>
#include <fcntl.h>
#include <setjmp.h>
#include <trap.h>
#include <io.h>
#include <txosvc.h>

int keyboard, card, bar, except_slot, normal_slot;
long seconds = 0;
jmp_buf main_loop; /* used with setjmp() and longjmp() */
```

```
char buffer [500];

/*********************/
/* Exception hanlder. */
/*********************/
void exception (trap_no) int trap_no;
{
    switch (trap_no) {
    case TRAP_KEY_CLEAR :
              printf("\fCLEAR");
              disarm_guard (except_slot);
              disarm_guard (normal_slot);
              seconds = 0L;
              close(keyboard);
              close(card);
              close(bar);
              longjmp (main_loop, 1);
    default :
              printf("\fUNKNOW ERROR");
              break;
    }
}

/*******************/
/* Normal handler. */
/*******************/
void normal (trap_no) int trap_no;
{
    int sz;
    switch (trap_no) {
    case TRAP_KEY_OTHER :
              read(keyboard, buffer, 1);
              printf("\fKEY=%2X HEX", buffer[0]);
              break;
    case TRAP_CLOCK_SEC :
              seconds++;
              if ((seconds % 10L) == 0)
                  printf("\f%ld MIN %ld SEC",
                         seconds / 60L, seconds % 60L);
              break;
    case TRAP_MAG_CARD_AVAIL :
              sz = read(card, buffer, sizeof(buffer));
              printf("\fCARD DATA SZ %d", sz);
```

```
                break;
        case TRAP_BAR_CODE_AVAIL :
                buffer[sz = read(bar, buffer,
                                 sizeof(buffer)-1)] = 0;
                printf("\f%s", buffer);
                break;
        }
}

/*******************/
/* Main program.   */
/*******************/
main ()
{
    struct trap_mask take;
    if (setjmp (main_loop) != 0) { /* Abort logic */
         SVC_WAIT(1000);
         printf("\fRESTART");
         SVC_WAIT(1000);
    }
    keyboard = open ("/dev/stdin",  O_RDONLY);
    card     = open ("/dev/card",   O_RDONLY);
    bar      = open ("/dev/bar",    O_RDONLY);

    /* Set up the exception handler */
    take = mask_of(TRAP_KEY_CLEAR);
    except_slot = arm_guard (exception, take, take);

    /* Set up the normal completion handler */
    take = mask_or(mask_of(TRAP_KEY_OTHER),
                   mask_or(mask_of(TRAP_CLOCK_SEC),
                           mask_or(mask_of(TRAP_MAG_CARD_AVAIL),
                                   mask_of(TRAP_BAR_CODE_AVAIL))));
    normal_slot = arm_guard(normal, take, take);

    printf("\fREADY");

    while (1) ;  /* main program does no flow control */
}
```

Exception Handling Example #3, USERTRAP.C

```
/* USERTRAP.C */

/*************************************************************/
/*                                                           */
/*  This program demonstrates user traps.                    */
/*                                                           */
/*  The program will divide two integers input by the user.  */
/*  Processing will stop if the user presses the CLEAR       */
/*  key during input or if the denominator is equal to 0.    */
/*                                                           */
/*************************************************************/

#include <stdio.h>
#include <trap.h>
#include <config.h>
#include <stdlib.h>

/* The following interpreter trap definitions are defined
   in <TRAP.H> in newer releases of TXO Workbench.        */

/* Four user defined traps. */
#define TRAP_USER4              28
#define TRAP_USER3              29
#define TRAP_USER2              30
#define TRAP_USER1              31

/*****************************************************************/

/*  Error message if divide by 0 occurs.
*/
/*****************************************************************/

void divide_error()
{
  printf("\fHALT-DIVIDE BY 0");
} /* divide_error */

/*****************************************************************/

/* Error message if user presses the CLEAR key to abort input.*/
/*****************************************************************/
```

```
void enter_error()
{
  printf("\fUSER ABORT");
} /* enter_error */

/*********************************************************************/

/* Get an integer. Abort the program if the CLEAR key is pressed.
*/
/*********************************************************************/

int get_int(min_len, max_len)
int min_len,
    max_len;
{
  char dest[16];
  int num_read;

  num_read = SVC_KEY_TXT(dest, 0, max_len, min_len);
  if (num_read == -1)
    {
    dispatch_guard(mask_of(TRAP_USER3));
    exit(2);
    }
  else
    {
    *(dest+num_read+1) = 0;
    return(atoi(dest+1));
    }
} /* get_int */

/*********************************************************************/

/* Divide two integers.  Abort if a divide by 0 attempt is made.
*/
/*********************************************************************/

int divit(numer, denom)
int numer,
    denom;
{
  if (denom == 0)
     {
     dispatch_guard(mask_of(TRAP_USER4));
```

```
      exit(1);
      }
   else
      return(numer / denom);
} /* divit */

main()
{
   int  div_slot,
        abort_slot;
   char numer,
        denom;
   struct trap_mask div_trap,
                    abort_trap;
/*
   This section sets up the divide_error() function to be invoked if
   there is a division by 0 error in the divit function.
*/
   div_trap = mask_of(TRAP_USER4);   /* TRAP_USER4 = 28 */
   div_slot = arm_guard(divide_error, div_trap, div_trap);

/*
   This section sets up the enter_error() function to be invoked if
   the user presses the CLEAR key during input.
*/
   abort_trap = mask_of(TRAP_USER3);     /* TRAP_USER5 = 29 */
   abort_slot = arm_guard(enter_error, abort_trap, abort_trap);

/*
   Get the numerator.
*/
   printf("\fNUMERATOR?");
   numer = get_int(1, 2);

/*
   Get the denominator.
*/
   printf("\fDENOMINATOR?");
   denom = get_int(1, 2);

   printf("\f%d", divit(numer, denom));
}
```

C H A P T E R

8

System Devices

This chapter describes each of the system devices:

- Keyboard
- Display
- Beeper
- Magnetic Card Reader
- Real Time Clock/Calendar

System devices are accessed in the same manner as files, using the same basic set of function calls: `open()`, `read()`, `write()`, `ioctl()`, `seek()`, and `close()`. Like files, system devices are specified by name, prefixed with "`/dev/`". For example, to open the clock device, the device name "`/dev/clock`" is used. Like file names, device names are not case sensitive.

The basic function calls always return an error code of -1 when an error condition is encountered, and set `errno` to a specific error code.

The names of the OMNI 300 Series terminal standard devices are defined in <io.h> and <config.h> and are listed below:

STDIN	Keyboard
STDOUT	Display
STDERR	Beeper
DEV_CLOCK	Real time clock/calendar
DEV_CARD	Magnetic card reader
DEV_BAR	Optical bar code reader
DEV_COM1	External RS-232 async host link
DEV_COM3	External RS-232 sync/async host link
DEV_COM4	Dial models: Internal 103/212/V.21/ V.22 modem LAN models: Peer-to-peer LAN
DEV_MODEM	Internal 103/212/V.21/V.22 modem
DEV_PINPAD	Personal ID entry device (COM2). Also used for the OMNI 460 internal printer.

Normally, all devices must be explicitly opened; however, the following logical device names may used to access a device without a specific open:

```
STDIN
STDOUT
STDERR
CLOCK
```

Keyboard

All OMNI 300 Series keyboards use a common 4x4 keypad consisting of a 12-key numeric keypad in either a telco- or calculator-style layout. OMNI 300 Series models with higher functionality also include "soft" keys that are application programmable. These keys are sometimes related to the terminal's display (screen addressable keys) or serve as "function" keys in a PC fashion. An application may assign any functionality to these keys

The keyboard buffer size is 20 characters (keystrokes). If no `read()`s are made after 20 keystrokes, any following keystrokes are lost.

Certain simultaneous key presses are also recognized and translated to a unique data byte called a "key code".

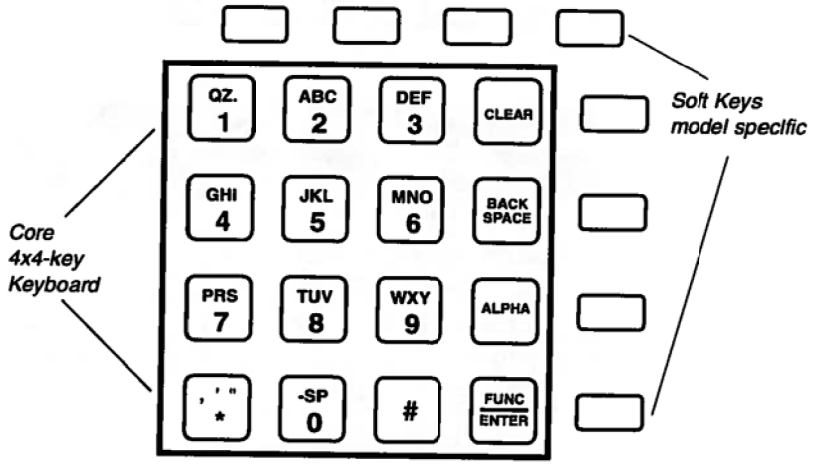

Figure 8-1. Typical OMNI 300 Keyboard Layout

4x4-key Core Keypad Key Coding

The core 4x4 keypad has a total of 16 keys as shown in Figure 8-1. Each key, when pressed, causes a unique key code (its ASCII equivalent) to be stored in a device-level buffer as shown in Table 8-1. Valid key combinations are also recognized, as listed in Table 8-2. Table 8-3 shows constants defined for various function keys. Invalid key presses produce unpredictable results.

Table 8-1. Keys and Key Codes

Key	Code	Key	Code	Key	Code
[0]	0x30	[5]	0x35	[CLEAR]	0x1B (ESC)
[1]	0x31	[6]	0x36	[ENTER]	0x0D (CR)
[2]	0x32	[7]	0x37	[ALPHA]	0x0E (SO)
[3]	0x33	[8]	0x38	[BKSP]	0x08 (BS)
[4]	0x34	[9]	0x39		

Table 8-2. Shifted Keys and Key Codes

Key Sequence	Key Codes
[ALPHA]+[0–6]	0xA0 - 0xA6
[BACKSPACE]+[0–3] [BACKSPACE]+[7–9]	0xB0 - 0xB3 0xB7 - 0xB9
[CLEAR]+[0–9]	0xC0 - 0xC9
[*]+[0–9]	0xD0 - 0xD9
[FUNC/ENTER]+[0-9]	0xE0 - 0xE9
[#]+[0-9]	0xF0 - 0xF9

Table 8-3. Keyboard Constants (defined in <io.h>

Key	Key Codes	Key	Key Codes
[CLEAR]	kbd_clear	[ENTER]	kbd_enter
[ALPHA]	kbd_alpha	[BACK SPACE]	kbd_bk_sp

Function Keys

When the user presses a function key, the system simply passes the key's code to the application, which acts upon the key press according to the application design. Four screen-addressable function keys are laid out just below the display, allowing the display and key to work jointly as a prompt-and-response key. The remaining four function keys are laid out beside the core keypad.

The following table shows the key codes for the eight programmable function keys found on the OMNI 390 and 395 terminals.

Table 8-4. OMNI 390 and 395 Programmable Function Keys

Screen-addressable Keys Left to Right		Vertical Function Keys Top to Bottom	
Key	**Code**	**Key**	**Code**
Functiona	0x61	**Functione**	0x65
Functionb	0x62	**Functionf**	0x66
Functionc	0x63	**Functiong**	0x67
Functiond	0x64	**Functionh**	0x68

Keyboard Function Calls

open()

Explicitly opens the keyboard device, clearing the keyboard buffer and returning a handle to the keyboard, which is used in all subsequent device calls.

```
#include <io.h>

hKBD = open ("/dev/stdin", 0);
int hKBD;
```

The keyboard does not need to be explicitly opened; the logical device name `STDIN` may be used by other I/O functions in place of the device handle `hKBD`.

read()

Transfers key code data from the internal keyboard buffer to the application's buffer.

```
#include <io.h>

bytes_read = read (hKBD, buffer, size);
int bytes_read, hKBD, size;
char *buffer;
```

The `size` parameter specifies the maximum number of bytes to be read. The size of the internal keyboard buffer is 20 bytes. Once the internal buffer is read, it is cleared for new keyboard input or for `write()` operations. If no keys have been pressed, `bytes_read` is zero.

Bytes are removed from the keyboard buffer when they are read. If not all bytes in the buffer are read, the remaining bytes are moved to the beginning of the buffer, allowing additional bytes to be appended to the keyboard buffer.

This function performs a "low-level" read: no translation of multiple key sequences is performed. For example, the key sequence [1] [ALPHA] [ALPHA] returns 3 bytes: 0x31, 0x0E and 0x0E rather than the letter "Z". Use the library function `SVC_KEY_TXT()` to provide key sequence translation.

write()

While normally thought of as a read-only device, the keyboard can be "written" to, emulating keystrokes entered by a user. Data written to the keyboard is also stored in the internal keyboard buffer.

```
#include <io.h>

bytes_written = write (STDIN, buffer,iCount);
int bytes_written, STDIN, iCount;
```

A maximum of 20 characters may be written to the keyboard buffer. Once this limit is reached, further write attempts will fail until the keyboard buffer is read.

When used with `arm_guard()`, writing to the keyboard triggers an event trap (Trap 32 or 33).

ioctl()

The keyboard supports several device control functions which use the `GetCtrl` and `SetCtrl` constants defined in the `<io.h>` header file.

The following function returns the number of characters in the internal keyboard buffer:

```
#include <io.h>

iResult = ioctl (STDIN, GetCtrl | 0, 0);
int iResult;
```

The following function sounds a short beep each time a key is pressed:

```
#include <io.h>

iResult = ioctl (STDIN, SetCtrl | 0, 0);
int iResult;
```

Use the following function to disable the short beep with each keystroke:

```
#include <io.h>

iResult = ioctl (STDIN, SetCtrl | 1, 0);
int iResult;
```

The default is beep enabled.

See SVC_INFO_KEY() for additional information on ioctl-type operations.

close()

Releases and terminates use of a keyboard handle. The keyboard remains active; scanning continues and key codes are still buffered, but the internal resources related to the handle are released. This function only needs to be used if the keyboard was specifically opened.

```
#include <io.h>

iStatus = close (STDIN);
int iStatus;
```

CLEAR Key Handling

A [CLEAR] key press always places an ESC (0x1B) value in the keyboard buffer. In addition, the system allows special handling of the [CLEAR] key through the standard guard mechanism described in Chapter 7, *Exception Handling*.

Keyboard Example

```
 /* KEYBOARD.C                                              */
 /* This program demonstrates keyboard functions. */

#include <stdio.h>
#include <io.h>
#include <memory.h>
#include <errno.h>
#include <txosvc.h>

main()
 {
     int kbd_handle,in_buffer;
     char i;
     char buffer [20];
     memset (buffer,0,20);

/* Open keyboard */
     kbd_handle = open("/dev/stdin",0);
     if (-1 == kbd_handle) printf ("\fERROR # %d",errno);

/* Turn off keyboard beeps */
     ioctl (kbd_handle,SetCtrl | 1,0);

/* Loop six times, getting one byte from the keyboard
   buffer each time. After the third loop, turn the
   keyboard beeps back on. */
     printf ("\fPRESS A KEY");
     for (i=0;i<6;++i)
     {
     /* READ() will return zero until a key is pressed */
     while (0 == read(kbd_handle,buffer,1));
```

Continued

```
      /* Turn keyboard beeps on */
      if (2 == i) ioctl (kbd_handle,SetCtrl | 0,0);
      printf ("\fKEY WAS %s",buffer);
      }
      SVC_WAIT(1000);

/* Write to the keyboard buffer */
      if (-1 == write (kbd_handle,"THIS IS A TEST",14))
           printf("\fERROR # %d",errno);

/* Get the number of bytes in the buffer */
      in_buffer=ioctl (kbd_handle,GetCtrl | 0,0);
      printf ("\f%d BYTES WAITING",in_buffer);
      SVC_WAIT (1000);

/* Read the keyboard buffer and print */
      if (-1 == read (kbd_handle,buffer,in_buffer))
           printf ("\fERROR # %d",errno);
      printf ("\f%s",buffer);
      SVC_WAIT (1000);

/* Close keyboard and check for error */
      if (0 == close(kbd_handle))
        printf ("\fKEYBOARD CLOSED");
        else printf ("\fERROR # %d",errno);
}
```

Segment-type Display

OMNI 300 Series terminals offer two types of model-dependent displays: a character-oriented, segment-type display, and a graphics-oriented pixel-type display. This section discusses the segment-type display.

Segment-type displays are so called because each character position uses a 14-segment vacuum fluorescent display tube grid to implement a subset of the ASCII character set consisting of the uppercase alphabet (A-Z), numeric characters (0-9), as well as certain special characters such as * , ' " - . # : ! + @ = & and a space.

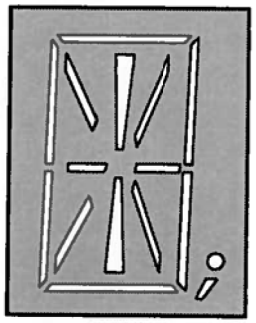

Figure 8-2. Segment Grid

Each 14-segment grid includes a period (or decimal) character and a comma character. Periods and commas are included with their preceding character and do not take up a character position unless they are the leftmost character in the string. For example, "100.00" uses only five character positions, as the period is on the segment grid showing the zero in the one's place.

All data written to the display is stored in a 32-character buffer. The display shows only 16 characters, alpha or numeric, at one time. Up to 32 characters in the buffer can be displayed if periods or commas are used on each segment grid. All data sent to the display is assumed to be 7-bit ASCII. Any character which the display device is unable to properly display will be shown as an underscore (ASCII 0x5F). To clear the display, send the form feed character (`'\f'` or 0x0C).

Segment Display Function Calls

open()

Clears the display and places the cursor in the home position (the first character position of the first line). A handle is returned for use in subsequent display operations.

```
#include <io.h>

hDSP = open ("/dev/stdout", 0);
int hDSP;
```

The display does not need to be explicitly opened; the logical device name STDOUT may be used by other I/O functions in place of the device handle hDSP.

write()

Transfers data from the application buffer to the internal display buffer, where it is shown in the current display window.

```
#include <io.h>

bytes_written = write (hDSP, buffer, size);
int  bytes_written, hDSP, size;
char *buffer;
```

This function writes size number of characters to the buffer and returns the actual number of bytes written.

read()

While the display is normally thought of as a write-only device, the display can be "read" from as well. This function returns the contents of the internal display buffer.

```
#include <io.h>

bytes_read = read (hDSP, buffer, size);
int bytes_read, hDSP, size);
char * buffer;
```

The `size` parameter specifies the maximum number of bytes to be read.

close()

Releases the handle associated with the display.

```
#include <io.h>

status = close (hDSP);
int status, hDSP;
```

Related Display Functions

In addition to the basic "low level" calls described earlier, many higher-level display functions are provided as well. A complete description of each of these functions is contained in *Chapter 11, Library Functions.*

`window()`	Defines a logical display window
`gotoxy()`	Positions cursor
`clrscr()`	Clears screen
`clreol()`	Clears display from current position to end of current line
`delline()`	Deletes line containing cursor
`insline()`	Inserts line below line containing the cursor
`wherecur()`	Returns cursor location relative to physical display
`wherewin()`	Returns current window coordinates
`wherewincur()`	Returns cursor location relative to current window
`setscrollmode()`	Controls display scrolling
`getscrollmode()`	Returns display scrolling method

Windows effectively exist "one at a time". That is, only the currently declared window exists (or the default window filling the display if none has been explicitly created). To switch to another window, you must create it with the `window()` command. To return to a previously used window, recreate it with another suitably written call to `window()`. All other windowing functions and attributes are relevant to whatever window is current. Thus, setting the scroll mode will effect all windows created until the scroll mode is explicitly changed.

Segment-type Display Example

```
/* DISPLAY.C */
 /* This program demonstrates the use of most window
    functions. */

#include <stdio.h>
#include <io.h>
#include <txostd.h>
#include <txosvc.h>

int dsp_handle,x,y,x1,y1,x2,y2;
char c,buffer[4];
main () {

   /* Open display. */
   dsp_handle=open("/dev/stdout", 0);

   /* Write "OMNI 380" to display and make "380" appear to
      blink by erasing, pausing, and rewriting "380" repeatedly. */
   write(dsp_handle, "OMNI 380", 8);
   for (x=0; x<10; x++) {
      gotoxy(6, 1);
      clreol();
      SVC_WAIT(75);
      write(dsp_handle, "380", 3);
      SVC_WAIT(150);
   }

   /* Treat the display as having four 4 character windows:

                 ____________________
                 |win1|win2|win3|win4|
                 --------------------

      Cycle through moving:

          win1 to win3
          win2 to win4
          win3 to win1
          win4 to win2

      Continue until a key is pressed. */
```

```
    x1=1;
    do {
       /* Define, read, then clear window to be moved. */
       window(x1, 1, x1+3, 1);
       read(dsp_handle, buffer, 4);
       clrscr();

       /* Define and write to receiving window. */
       x1=(x1+8) % 16;
       window(x1, 1, x1+3, 1);
       write(dsp_handle, buffer, 4);
       SVC_WAIT(1000);

       /* Set x1 to start of next window to move. */
       wherecur(&x, &y);
       x1=(x+8) % 16;

    } while (read(STDIN, &c, 1) == 0);

    /* Display "HERE" in the last window that was defined above.*/
    wherewin(&x1, &y1, &x2, &y2);
    window(1, 1, 16, 1);
    printf("\fLAST WINDOW WAS");
    SVC_WAIT(1000);
    delline();
    window(x1, y1, x2, y2);
    write(dsp_handle, "HERE", 4);

    /* Close the display. */
    close(dsp_handle);
}
```