# put_env()

Stores an environment variable in CONFIG.SYS.

**Prototype**

```
#include <txostd.h>
int put_env( key, buffer, count );
char *key, *buffer;
int count;
```

**Parameters** `key` is a null-terminated string containing the name of the environment variable, `buffer` is a pointer to an array containing the environment variable data, and `count` specifies the size of the buffer.

**Returns**

| | |
|---|---|
| 0 | Success. |
| -1 | The key does not exist or the operation failed, with `errno` set to a specific error value. |

**Notes** When existing key data is deleted, `count` is set to 0.

**Related** `get_env()`

**Example**

```
/* PUT_ENV.C */

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

char buffer[10];
main () {

   /* Create a new variable in CONFIG.SYS */
   put_env("NEWKEY", "NEWVAL", 6);

   /* Get the value of 'NEWKEY'.
      Use return length to make NULL-terminated string. */
   buffer[get_env("NEWKEY", buffer, 10)]=0;

   /* Display value of 'NEWKEY' */
   printf("\fNEWKEY = %s",buffer);
}
```

# putkey()

Stores the data associated with a given key into a file.

**Prototype**

```
#include <txostd.h>
int putkey( key, buffer, count, file_name );
char *key, *buffer, *file_name;
int count;
```

**Parameters** `file_name` is a pointer to a null-terminated string identifying the file, `key` is a null-terminated string containing the key data, `buffer` is a pointer to an array containing the data, and `count` specifies the size of the buffer.

**Returns** If successful, no meaningful value is returned.

If an error occurs, `-1` is returned with `errno` set to `EBADF`.

**Notes** The keyed file does not have to be open; an implied open and close is performed automatically. `putkey()` can be used successfully even when the allowable number of open files has been reached. However, `file_name` must reference an existing file created by the `open()` function.

**Related** `getkey()`

**Example** See following page.

### Example

```
/* KEY.C */

#include <stdio.h>
#include <fcntl.h>
#include <txostd.h>

char buffer[10];
int file_handle;
main () {

   /* File must exist, but doesn't need to be open. */
   file_handle=open("NEWFILE",O_CREAT);
   close(file_handle);

   /* Create a new variable in NEWFILE */
   putkey("KEY", "VALUE", 5, "NEWFILE");

   /* Get the value of KEY
      Use return code to make NULL-terminated string. */
   buffer[getkey("KEY", buffer, 10, "NEWFILE")]=0;

   /* Display the value of KEY */
   printf("\fKEY = %s",buffer);
}
```

# put_lan_config()

Configures the LAN with parameters supplied by application and the CONFIG.SYS file.

**Prototype**

```
#include <lan.h>
int put_lan_config (handle, term_parms);
int handle;
LAN_TERM_PARMS *term_parms;
```

**Parameters**

`handle` is the value returned by the `open()`, and `term_parms` is a data structure containing the application-related configuration parameters for the LAN.

**Returns**

| | |
|---|---|
| 0 | Success |
| -1 | Failure, with errno set to a specific error value (if unable to configure the LAN, or if `*LAD` is not set in CONFIG.SYS). |

**Notes**

The data structure `LAN_TERM_PARMS` (in `<lan.h>`) is as follows:

```
typedef struct
{       unsigned char receive_timeout;
        unsigned char resource_definition;
        unsigned char transmit_retries;
        unsigned char throttle;
} LAN_TERM_PARMS;
```

`receive_timeout` specifies the number of seconds (1–10) that the terminal waits for an ACK before the transmission attempt fails.

`resource_definition` is for application use, e.g., to identify specific LAN resources, such as printer server or host gateway.

`transmit_retries` specifies the number of additional attempts (1 to 10) to transmit an unACKed message, e.g., `transmit_retries=3` means a total of four tries.

`throttle` is usually on (1) in most terminals on the LAN, preventing low address terminals from hogging the LAN under heavy loads, and off (0) for terminals needing better LAN throughput (gateway or server terminals).

**Related** `get_lan_config()`

**Example**

```
/* PUTLCFG.C */

#include <lan.h>
LAN_TERM_PARMS term_parms;
main()
{
   /* Set the LAN terminal parameters. */
   term_parms.receive_timeout = 3;
   term_parms.resource_definition = 1; /* Print server. */
   term_parms.transmit_retries = 3;
   term_parms.throttle = 0; /* Off for faster throughput.  */

   /* Configure the LAN. */
   result = put_lan_config (hLAN, &term_parms);

}
```

390 DIAL
395 LAN
395 DIAL

# putpixelcol()

Displays graphic images on a byte-by-byte basis.

**Prototype**

```
void putpixelcol(buffer,length);

char *buffer;
int   len;
```

**Parameters** `buffer` contains display graphics, `length` specifies the buffer length (in bytes).

**Returns** VOID

**Notes** Available only to pixel-type display terminals.

This function writes graphic display information to the display on a byte-by-byte basis, starting at the current cursor position. Each byte in the buffer represents the on/off state of a column of 8 pixels, where 1 means the pixel is on (darkened) and 0 does nothing to the display. Note that this function OR's the pixel state to the current display state (if the pixel is on it won't turn it off), so the application may want to clear the display area before calling this function.

The buffer is written by columns from the top of the character cell to the bottom moving from left to right. Once an entire character cell in the given font is filled, the cursor position is incremented.

Note that different grid settings take varying byte amounts to fill one character cell. For example, the 2x18 grid setting requires 16 bytes to fill one character, the 3x18 grid requires 10 bytes and the 4x25 grid requires 6 bytes.

This function works on the current window, not the entire display.

Data for this function can be created by the VeriFone PC program FontDesigner, and is linked into the application (or downloaded as a separate data file). Data created by FontDesigner specifies a single grid setting. In order for the image to properly appear, the application must set the grid to the proper setting before calling `putpixelcol()`.

**Example** See following page.

**Example**

```
/* PUTPIXEL.C */

#include <txostd.h>
#include <dsp_90.h>

void main()
{
   char buffer[8];

   buffer[0] = (char)0xFF;
   buffer[1] = (char)0x42;
   buffer[2] = (char)0x24;
   buffer[3] = (char)0x18;
   buffer[4] = (char)0x18;
   buffer[5] = (char)0x24;
   buffer[6] = (char)0x42;
   buffer[7] = (char)0x81;

   clrscr();

   putpixelcol(buffer, 8);
}
```

# read()

Transfers data from a file or device to the application's buffer area.

**Prototype**

```
#include <io.h>
int read( handle, buffer, count );
int handle, count;
char *buffer;
```

**Parameters**

`handle` is the value returned by `open()`.

`count` is the number of bytes to be transferred.

`buffer` is a pointer to a region of memory (must be large enough to hold `count` bytes).

**Returns**

| | |
|---|---|
| `bytes_read` | The number of bytes read on a successful read; |
| 0 | An attempt was made to read at end-of-file; |
| -1 | Failure, with `errno` set to a specific error value. |

**Notes**

The result of the read operation will vary, depending on the data source, either from a file or from a device.

**Example**

See following page.

### Example

```
/* FILE.C */
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
int file_handle;
char buffer[20];
main () {

   /* Create file for read and write. */
   file_handle = open("Test.Dat", O_CREAT+O_RDWR);

   /* Write to the file. */
   write(file_handle, "Hello World", 11);

   /* Close the file */
   close(file_handle);

   /* Open the file for read only. */
   file_handle = open("Test.Dat", O_RDONLY);

   /* Read from the file. */
   read(file_handle, buffer, 11);

   /* Display the contents of the buffer. */
   buffer[11]=0;
   printf("\f%s",buffer);

   /* Close the file */
   close(file_handle);
}
```

# read_cmd()

Reads a modem command response.

**Prototype**

```
#include <device.h>
int read_cmd( modem_handle, buffer, size );
int bytes_read, modem_handle, size;
char *buffer;
```

**Parameters**

`modem_handle` is the value returned from `open()`.

`size` is the maximum number of bytes to be read.

`buffer` is a pointer to the data area (must be large enough to hold `size` bytes).

**Returns**

| | |
|---|---|
| `bytes_read` | Each `read_cmd()` call returns the number of bytes actually read. |
| 0 | The modem has not sent a response. |

**Notes**

Typically, an application calls `read_cmd()` after every `write_cmd()` to check the modem's response.

Opening the modem device results in two initialization responses that should be checked with `read_cmd()`: one checks the modem's default parameters, and one checks for any user parameters set in the *MI entry in CONFIG.SYS.

**Related**

`write_cmd()`

**Example**

See following page.

**Example**

```
/* MODEMCMD.C */
#include <stdio.h>
#include <device.h>
int modem,bytes;
char buffer[10];
struct Opn_Blk opnblk;

main() {

   /* Open the modem. */
   modem = open("/dev/com4",0);

   /* Read past the two responses resulting from
opening */
   /* the modem.  Each response should be 2 bytes
long.  */
    for (bytes=0; bytes<4;)
       bytes += read_cmd(modem, &buffer[bytes], 2);

   /* Set up the open block and initialize using ioctl. */
   opnblk.rate=Rt_1200;
   opnblk.format=Fmt_A7E1;
   opnblk.protocol=P_pakt_mode;
   opnblk.parameter=0;
   opnblk.trailer.packet_parms.stx_char=2;
   opnblk.trailer.packet_parms.etx_char=3;
   opnblk.trailer.packet_parms.count=1;
   if (!ioctl(modem,0,&opnblk)) {
      printf("\fIOCTL SUCCESSFUL");
      getchar();
   }

   /* Write/Read a successful command (Set BELL
Mode) */
   write_cmd(modem,"ATB1\015",5);
   while(!read_cmd(modem,buffer,10));
```

# read_cvlr()

Reads a compressed variable-length record from a file.

**Prototype**

```
#include <vlr.h>
int read_cvlr( handle, buffer, size );
int handle, size;
char *buffer;
```

**Parameters**

`handle` is the value returned by `open()`.

`size` is the maximum number of bytes to read from the record, starting at the current file position indicator.

`buffer` is the memory area that the data from the record should be copied into. It must be large enough to hold `size` bytes.

**Returns**

| | |
|---|---|
| `bytes_placed` | The number of uncompressed bytes placed into the buffer. |
| 0 | An attempt was made to read at the end of the file. |
| -1 | Failure, with `errno` set to a specific error value. |

**Notes**

After reading a record, the file position pointer is advanced to the start of the next record (next data byte after the record) regardless of the number of bytes actually transferred to the application's buffer.

Although files normally contain only one type of data, they may contain a mixture of binary data, normal variable-length records, and compressed variable-length records. For this function to read the record successfully, the file position pointer must be positioned at the beginning of a compressed variable-length record.

**Example**

See following page.

**Example**

```
/* CVLR.C */

#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <vlr.h>
int file_handle;
char buffer[20];
main () {

   /* Create file with read and write. */
   file_handle = open("Test.Dat", O_CREAT+O_RDWR);

   /* Write compressed variable length records to the file. */
   write_cvlr(file_handle, "Hello ", 6);
   write_cvlr(file_handle, "Small ", 6);
   write_cvlr(file_handle, "World", 5);

   /* Seek to the beginning of the word "Small" */
   seek_cvlr(file_handle, 1, SEEK_SET);

   /* Delete "Small " */
   delete_cvlr(file_handle, 1);

   /* Insert "Big" */
   insert_cvlr(file_handle, "Big ", 4);

   /* Read from the file and display the result. */
   seek_cvlr(file_handle, 0, SEEK_SET);
   read_cvlr(file_handle, buffer, 6);
   read_cvlr(file_handle, buffer+6, 4);
   read_cvlr(file_handle, buffer+10, 5);
   buffer[15]=0;
   printf("\f%s",buffer);

   /* Close the file */
   close(file_handle);
}
```

# read_reject_pkt()

Reads and retrieves a rejected message from the reject queue. Only for use with the VISA First Generation protocol (using modem on COM1), or with a LAN message.

**Prototype**

```
#include <device.h>
int read_reject_pkt( handle, buffer, size );
int handle, size;
char *buffer;
```

**Parameters** `handle` is the value returned by `open()`, `buffer` is the pointer to a region of memory to receive the rejected message, with `size` (bytes) specifying the buffer's length.

**Returns**

| | |
|---|---|
| `bytes_placed` | The number of bytes of the rejected message actually read. If successful, `buffer` will then contain the next reject message from the queue (which will have also been deleted from the reject queue). |
| 0 | Reject queue is empty. |
| -1 | Failure, with `errno` set to a specific error value. |

**Notes** If `size` is smaller than the actual number of bytes in the rejected message, only the number of bytes specified by `size` will be placed in the buffer; extra characters will be deleted and lost.

**Example** See following page.

### Example

```
/* READREJ.C */

#include <lan.h>
extern int lan_handle;
xmit_failed () /* This routine is called when the XMIT_FAIL-
URE trap is detected. */
{
   char buffer[1518];
   int bytes_read;

   /* Read the next reject from the queue. */
   bytes_read = read_reject_pkt (lan_handle, &buffer,
sizeof(buffer));

   /* Call routine to post the item to a local batch. */
   post_local (&buffer, bytes_read);

  /* Warn operator. */
  printf("\fLAN XMIT FAILED.");

  return ();
}
```

# read_vlr()

Reads a variable-length record from a file.

**Prototype**

```
#include <vlr.h>
int read_vlr( handle, buffer, size );
int handle, size;
char *buffer;
```

**Parameters**

`handle` is the value returned by `open()`.

`size` is the maximum number of bytes to read from the record, starting at the current file position indicator.

`buffer` is the memory area that the data from the record should be copied into. It must be large enough to hold `size` bytes.

**Returns**

| | |
|---|---|
| `bytes_read` | The number of bytes read on a successful read. |
| `0` | An attempt was made to read at the end of the file. |
| `-1` | Failure, with `errno` set to a specific error value. |

**Notes**

After reading a record, the file position pointer is advanced to the start of the next record (next data byte after the record) regardless of the number of bytes actually transferred to the application's buffer.

Although files normally contain only one type of data, they may contain a mixture of binary data, normal variable-length records, and compressed variable-length records. For this function to read the record successfully, the file position pointer may be positioned at the beginning of a normal variable-length record.

**Example**

See following page.

**Example**

```
/* VLR.C */

#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <vlr.h>
int file_handle;
char buffer[20];
main () {

   /* Create file with read and write. */
   file_handle = open("Test.Dat", O_CREAT+O_RDWR);

   /* Write variable length records to the file. */
   write_vlr(file_handle, "Hello ", 6);
   write_vlr(file_handle, "Small ", 6);
   write_vlr(file_handle, "World", 5);

   /* Seek to the beginning of the word "Small" */
   seek_vlr(file_handle, 1, SEEK_SET);

   /* Delete "Small " */
   delete_vlr(file_handle, 1);

   /* Insert "Big" */
   insert_vlr(file_handle, "Big ", 4);

   /* Read from the file and display the result. */
   seek_vlr(file_handle, 0, SEEK_SET);
   read_vlr(file_handle, buffer, 6);
   read_vlr(file_handle, buffer+6, 4);
   read_vlr(file_handle, buffer+10, 5);
   buffer[15]=0;
   printf("\f%s",buffer);

   /* Close the file */
   close(file_handle);
}
```

390 DIAL
395 LAN
395 DIAL

# resetdisplay()

Sets the font and grid for a pixel-type display.

**Prototype**

```
#include <dsp_90.h>
int resetdisplay(font, grid_id);
char *font;
int grid_id;
```

**Parameters**

`font` is the name of the Font Definition File used for the font.

`grid_id` is a constant defined in `<device.h>` specifying the grid setting to use.

**Returns**

| | |
|---|---|
| 0 | Font and grid successfully set. |
| -1 | Error. Font and grid remain unchanged. |

**Notes**

Available only on pixel-type display terminals.

Font and grid settings are discussed in *Chapter 8, System Devices, Pixel-type Display*.

If the specified values are valid, the display and internal display buffer are cleared, the window is reset to the default window and the cursor is reset to the home position. The scrolling mode and current contrast setting remain unchanged.

**Example**

```
/* RESETDSP.C */

#include <stdio.h>
#include <dsp_90.h>

void main()
{

   resetdisplay("DEFAULT", 0);

   printf("\fDEFAULT FONT");
}
```

# seek_cvlr()

Sets the file position pointer to a specified record address within a file.

**Prototype**

```
#include <vlr.h>
long seek_cvlr( handle, offset, origin );
int handle, origin;
long offset;
```

**Parameters**

`handle` is the file handle returned by `open()`.

`offset` is the number of records to seek past.

`origin` is one of the following:

`SEEK_SET` – Seek from the beginning of the file.
`SEEK_CUR` – Seek from the current file position pointer.
`SEEK_END` – Seek from end of the file. (Use to append to the file.)

**Returns**

| | |
|---|---|
| `byte_address` | If successful, the absolute offset (byte address) is returned. |
| `-1` | Failure, with `errno` set to a specific error value. Values for `errno` include: `EBADF` `EINVAL` |

**Notes**

Seeking backwards is not supported by `seek_clvr()` and `seek_vlr()`; `SEEK_END` moves the pointer to the end of the file to append a record.

Although files normally contain only one type of data, they may contain a mixture of binary data, normal variable-length records, and compressed variable-length records. For this function to seek past the right quantity of data, the file position indicator must be positioned at the beginning of a compressed variable-length record.

See the example on the following page.

**Related**

`lseek()`, `seek_vlr()`

**Example**

```
/* CVLR.C */

#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <vlr.h>
int file_handle;
char buffer[20];
main () {

   /* Create file with read and write. */
   file_handle = open("Test.Dat", O_CREAT+O_RDWR);

   /* Write compressed variable length records to the file. */
   write_cvlr(file_handle, "Hello ", 6);
   write_cvlr(file_handle, "Small ", 6);
   write_cvlr(file_handle, "World", 5);

   /* Seek to the beginning of the word "Small" */
   seek_cvlr(file_handle, 1, SEEK_SET);

   /* Delete "Small " */
   delete_cvlr(file_handle, 1);

   /* Insert "Big" */
   insert_cvlr(file_handle, "Big ", 4);

   /* Read from the file and display the result. */
   seek_cvlr(file_handle, 0, SEEK_SET);
   read_cvlr(file_handle, buffer, 6);
   read_cvlr(file_handle, buffer+6, 4);
   read_cvlr(file_handle, buffer+10, 5);
   buffer[15]=0;
   printf("\f%s",buffer);

   /* Close the file */
   close(file_handle);
}
```

# seek_vlr()

Sets the file position pointer to a specified record address within a file.

**Prototype**

```
#include <vlr.h>
long seek_vlr( handle, offset, origin );
int handle, origin;
long offset;
```

**Parameters**

`handle` is the file handle returned by `open()`.

`offset` is the number of records to seek past.

`origin` is one of the following:

`SEEK_SET` – Seek from the beginning of the file.
`SEEK_CUR` – Seek from the current file position pointer.
`SEEK_END` – Seek from end of the file (for appends).

**Returns**

| | |
|---|---|
| `byte_address` | If successful, the absolute offset (byte address) is returned. |
| `-1` | Failure, with `errno` set to a specific error value. Values for `errno` include: `EBADF` `EINVAL` |

**Notes**

Seeking backwards is not supported by `seek_clvr()` and `seek_vlr()`; `SEEK_END` moves the pointer to the end of the file to append a record.

Although files normally contain only one type of data, they are allowed to contain a mixture of binary data, normal variable-length records, and compressed variable-length records. For this function to seek past the right quantity of data, it is important for the file position indicator to initially be positioned at the beginning of a variable-length record.

See the example on the following page.

**Related**

`lseek()`, `seek_cvlr()`

**Example**

```
/* VLR.C */

#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <vlr.h>
int file_handle;
char buffer[20];
main () {

   /* Create file with read and write. */
   file_handle = open("Test.Dat", O_CREAT+O_RDWR);

   /* Write variable length records to the file. */
   write_vlr(file_handle, "Hello ", 6);
   write_vlr(file_handle, "Small ", 6);
   write_vlr(file_handle, "World", 5);

   /* Seek to the beginning of the word "Small" */
   seek_vlr(file_handle, 1, SEEK_SET);

   /* Delete "Small " */
   delete_vlr(file_handle, 1);

   /* Insert "Big" */
   insert_vlr(file_handle, "Big ", 4);

   /* Read from the file and display the result. */
   seek_vlr(file_handle, 0, SEEK_SET);
   read_vlr(file_handle, buffer, 6);
   read_vlr(file_handle, buffer+6, 4);
   read_vlr(file_handle, buffer+10, 5);
   buffer[15]=0;
   printf("\f%s",buffer);

   /* Close the file */
   close(file_handle);
}
```

390 DIAL
395 LAN
395 DIAL

# setcontrast()

Sets pixel-type display contrast level.

**Prototype**

```
#include <dsp_90.h>
void setcontrast(level);
int  level;
```

**Parameters** `level` ranges from 0 to 15, with 0 showing the most contrast, 15 showing the least contrast.

**Returns** VOID

**Notes** Available only to pixel-type display terminals.

This function sets the contrast level for the entire screen to the level specified. Values of level range from 0 to 15, with 0 having the most contrast and 15 the least. Lower contrast settings (higher parameter values) may cause the display to be unreadable by the terminal user.

**Related** `getcontrast()`

**Example**

```
/*  CONTRAST.C  */

#include <stdio.h>
#include <txosvc.h>
#include <dsp_90.h>

char buffer[10];
int contrast;
main () {

   while(1) {
      setcontrast(0);
      do {
         printf("\f  Enter Contrast\n(Between 0 and 15)");
         SVC_KEY_TXT(buffer, 0, 2, 1);
         contrast = SVC_2INT(buffer);
      } while ( (contrast < 0) || (contrast > 15));
      setcontrast(contrast);
      contrast=getcontrast();
      printf("\fThe Contrast is %2d  CLEAR To Reset", contrast);
      getchar();
   }
}
```

# set_env_buffer()

Sets the buffer used for returning environment variables.

**Prototype**

```
#include <txostd.h>
void set_env_buffer( bufptr, buffsize );
char *bufptr;
unsigned int buffsize;
```

**Parameters**

`bufptr` is a pointer to a user-allocated (or static) buffer to be used in all subsequent calls to the `getenv()` function.

`buffsize` is the actual size of the buffer.

**Returns**

VOID

**Notes**

The `getenv()` function normally allocates its buffers dynamically, and requires the application to free them (return them to the heap) after use. This function causes a user-provided buffer to be used for all subsequent calls to `getenv()`.

To revert to `getenv()` calls dynamically allocating buffers for return data, set `bufptr` to `NULL`.

**Related**

`getenv()`

**Example**

```
/* ENV.C */

#include <stdio.h>
#include <stdlib.h>
#include <txostd.h>

char p[20];
main () {

   /* Assign the environment variable "TEST" */
   put_env("TEST", "HELLO_WORLD", 11);

   /* Set the buffer for 'getenv' to write into. */
   set_env_buffer(p,20);

   /* Put the value of "TEST" into 'p' */
   getenv("TEST");

   /* Display the contents of 'p' */
   printf("\fp = %s",p);
}
```

390 DIAL
395 LAN
395 DIAL

# setfont()

Selects a Font Definition File (or default font) for a pixel-type display.

**Prototype**

```
#include <dsp_90.h>
int setfont(char *font);
char *font;
```

**Parameters**

`font` is a null-terminated font name.

**Returns**

| | |
|---|---|
| 0 | Font successfully changed. |
| -1 | Error. Font is invalid (check grid setting and font availability) or not supported by the firmware. |

**Notes**

Available only to pixel-type display terminals.

This function will change from the current Font Definition File to that pointed to by the `font` parameter (if the file is valid for the current grid setting and is available in the file system).

The `font` parameter may be either the file name of the desired Font Definition File, or the keyword `DEFAULT`, in which case the terminal's default font (ASCII 2 line by 18 character font) is loaded from the terminal's firmware.

The cursor position and current window setting are not changed by this function. The display and the internal display buffer are also not affected. The code size will change to the code size required by the new font.

See *Chapter 8, System Devices, Pixel-type Display* for more information on Font Definition Files and font code size.

**Related**

`getfont()`

**Example**

```
/* FONT.C */

#include <stdio.h>
#include <dsp_90.h>

void main()
{
   char font[13];

   getfont(font);
   printf("\fFONT: %s", font);

   setfont(font);
}
```

# setscrollmode()

Sets the display scrolling mode.

**Prototype**

```
#include <txostd.h>
void setscrollmode (mode);
int mode;
```

**Parameters** `mode` determines the manner in which the display is scrolled, as follows:

| | |
|---|---|
| `mode=0` | *Disable scrolling.* A write of a string larger than the size of the current window causes the leftmost part of the string to be displayed. |
| `mode=1` | *Enable horizontal scrolling—default.* A write of a string larger than the size of the current window causes the leftmost part of the string to scroll off the left side of the screen and display the rightmost characters. |
| `mode=2` | *Enable vertical scrolling.* A write of a string larger than the size of the current window causes the top line of the string to scroll off the top of the screen and the remaining lines to move up one line. |

**Returns** VOID

**Related** `getscrollmode()`

### Example

```
/* SCROLL.C */

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

main () {

   /* Fill display. */
   printf("\f>>>>>>>><<<<<<<<");

   /* Define window and move cursor to the last position. */
   window(3, 1, 14, 1);

   /* Display in the default mode, rightmost portion should remain. */
   /* Note that the default mode is identical to mode 1.          */
   /* >>NT MODE IS 1<< */
   printf("\fTHE CURRENT MODE IS %d", getscrollmode());
   getchar();

   /* Mode 0: Should not slide off the left edge of the window. */
   setscrollmode(0);
   /* >>0 IS THE CUR<< */
   printf("\f%d IS THE CURRENT MODE", getscrollmode());
   getchar();

   /* Mode 1: Should slide off the left edge of the window. */
   setscrollmode(1);
   /* >>NT MODE IS 1<< */
   printf("\fTHE CURRENT MODE IS %d", getscrollmode());
   getchar();

   /* Mode 2: When a character is written beyond the edge of the */
   /*         window, the current lines are scrolled up one line */
   /*         and the cursor is moved to the beginning of the    */
   /*         last line in the window.                           */
   setscrollmode(2);
   /* >>MODE IS 2   << */
   printf("\fTHE CURRENT MODE IS %d", getscrollmode());
}
```

# sound()

Generates specified beeper tones (notes) for a specified duration.

**Prototype**

```
#include <notes.h>
void sound( note, duration );
int note, duration;
```

**Parameters** The parameters specify a single tone using a `note` value to indicate the tone's pitch (see table below), and a `duration` value (in milliseconds) to indicate the length of the tone.

**Returns** VOID

**Notes** The note played is specified by a constant derived from the following table of supported notes, where the scale letter is given as the musical note followed by the octave number (e.g., `NOTE_C5`, middle C). Supported notes are indicated by a musical note.

| NOTE_ | C | Db | D | Eb | E | F | Gb | G | Ab | A | Bb | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | | | |
| 2 | | | | | | | | • | • | • | • | • |
| 3 | • | • | • | • | • | • | • | • | • | • | • | • |
| 4 | • | • | • | • | • | • | • | • | • | • | • | • |
| 5 | • | • | • | • | • | • | • | • | • | • | • | • |
| 6 | • | • | • | • | • | • | • | • | • | • | • | • |
| 7 | • | • | • | • | • | • | • | • | • | • | • | • |
| 8 | • | • | • | • | • | • | • | • | • | • | • | • |
| 9 | • | • | • | • | • | • | • | • | • | | | |

To use the above table, determine which note to play and enter the note (top row) and the octave (left column) following "`NOTE_`" in the code. Example: `NOTE_Db3` = D flat in the third octave. Notes range from G in the second octave and A flat in the ninth octave. Most notes useful for musical tones are in the middle ranges. Middle C is `NOTE_C5`, which generates a sound of approximately 523 Hz.

The constant NOTE_PAUSE specifies silence for duration number of milliseconds. These constants are defined in <notes.h>.

The maximum duration depends on the note being played, ranging from 30 seconds (NOTE_G2) to approximately 2 seconds (NOTE_Ab9). If the duration of a specified note exceeds the maximum allowed duration for that note, the system plays the note at its maximum duration.

Due to hardware limitations, the exact frequency played may be off by as much as a quarter step. This may result in some notes being noticeably out of tune. Values in NOTE_A5 through NOTE_C8 seem to work best.

**Example**

```
/* SOUND.C */
 (), programming example
#include <notes.h>

main () {

   /* Play beginning of
      'When the Saints Go Marching In' */
   sound(NOTE_C6, 400);   /* Oh      */
   sound(NOTE_E6, 400);   /* when    */
   sound(NOTE_F6, 400);   /* the     */
   sound(NOTE_G6, 1600);  /* saints  */
   sound(NOTE_PAUSE, 100);
   sound(NOTE_C6, 400);   /* Oh      */
   sound(NOTE_E6, 400);   /* when    */
   sound(NOTE_F6, 400);   /* the     */
   sound(NOTE_G6, 1600);  /* saints  */
   sound(NOTE_PAUSE, 100);
   sound(NOTE_C6, 400);   /* Oh      */
   sound(NOTE_E6, 400);   /* when    */
   sound(NOTE_F6, 400);   /* the     */
   sound(NOTE_G6, 800);   /* saints  */
   sound(NOTE_E6, 800);   /* go      */
   sound(NOTE_C6, 800);   /* mar-    */
   sound(NOTE_E6, 800);   /* ching   */
   sound(NOTE_D6, 1600);  /* in      */

}
```

# SVC_CHECKFILE()

Checks that the current computed checksum for a specified file matches the checksum value stored in the file directory.

**Prototype**

```
#include <txosvc.h>
int SVC_CHECKFILE (filename);
char *filename;
```

**Parameters** The `filename` parameter is null-terminated string identifying the file to be validated.

**Returns**

| | |
|---|---|
| 0 | File checksum is OK. |
| -1 | File checksum is bad. |
| -2 | File not found. |

**Notes** The system file DATA.EM will always appear to have a checksum error because it is not updated in the same manner as the other files. This error should be ignored.

This function is supported on EPROM Version 10 and higher.

### Example

```
/* SVC_CHECKFILE example */

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

char file[33];
int file_stat;
main() {

   /* Get the first file in the file system. */
   dir_get_first(file);

   /* Loop through each file. */
   do {
       printf("\f%s ", file);

       /* Check the file for checksum errors.                    */
       /* DATA.EM will always appear to have a checksum error. */
       file_stat = SVC_CHECKFILE(file);
       if (file_stat)
          printf("ERROR %d", file_stat);

       else
          printf("OK");
       SVC_WAIT(2000);

     } while (-1 != dir_get_next(file)); /* Get the next file. */
}
```

# SVC_CHK_PASSWORD()

Compares a counted string to the current system password.

**Prototype**

```
#include <txosvc.h>
int SVC_CHK_PASSWORD( buffer );
char *buffer;
```

**Parameters** `buffer` is a counted string containing the password to be compared to the system password.

**Returns**

1 `buffer` matches the system password.

0 `buffer` does not match the system password.

**Note** This function is supported on EPROM Version 10 and higher.

**Example**

```
/* SVC_CHK_PASSWORD example */

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

char buffer[100];
main () {

   /* Loop forever. */
   while (1) {

      /* The user will enter a possible password. */
      /* buffer will receive a counted string.    */
      printf("\fENTER PASSWORD");
      SVC_KEY_TXT(buffer, 1, 99, 1);

      /* 1 = valid password;  0 = invalid password */
      switch (SVC_CHK_PASSWORD(buffer)) {
         case 1: printf("\fVALID PASSWORD");   break;
         case 0: printf("\fINVALID PASSWORD"); break;

         default: printf("\fERROR BAD RESULT"); while(1);
      }

      SVC_WAIT (2000);
   }
```

# SVC_CLOCK()

Allows the user to read or set the current time.

**Prototype**

```
#include <txosvc.h>
int SVC_CLOCK( action, buffer, limit );
int action, limit;
char *buffer;
```

**Parameters**

`action`

0 = put the current time into `buffer` (reads clock).

1 = write the contents of `buffer` to the clock (sets clock).

`limit` indicates the maximum number of characters: 15 when reading the clock, 14 when writing to the clock.

**Returns**

| | |
|---|---|
| `bytes` | The number of bytes read or written. |
| `-1` | Failure. |

**Note** This function is supported on EPROM Version 10 and higher.

**Example** See following page.

**Example**

```
/* SVCCLOCK.C */

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

/*              yyyymmddhhmmssd */
char time[] = "20000101000000d";
char save[20], buffer[20];
int bytes;

main () {

    /* Save the current time. */
    bytes = SVC_CLOCK(0, save, 15);

    /* Change the time. */
    bytes = SVC_CLOCK(1, time, 14);

    /* Get and display the new time. */
    bytes = SVC_CLOCK(0, buffer, 15);
    clrscr();
    write(STDOUT, buffer, bytes);

    /* Restore and display the current time. */
    bytes = SVC_CLOCK(1, save, 14);
    getchar();
    bytes = SVC_CLOCK(0, buffer, 15);
    clrscr();
    write(STDOUT, buffer, bytes);
}
```