Keyed Files

Keyed files (also called paired files) are essentially CVLR files that use two records for every data record written. The first record, called a key, is followed by a data record. The key gives the data record an alphanumeric name or identifier, thus providing random access to the records.

Keyed file access is considerably slower than VLR or CVLR files, as two records must be read for each data entry, and a comparison of the key must be performed. From a timing performance perspective, keyed files are the least desirable.

Keyed files do have appropriate uses, however. Since keyed files can be edited via the terminal's keypad, they are ideal for holding data requiring modification, such as download parameters (telephone numbers and terminal ID) and customer specific information, such as a customer address.

The CONFIG.SYS file is a keyed file that is used to download parameters, and can be accessed from the system mode using the `SVC_STORE()` function.

Keyed files can also be used to create a database in which information could be located by a key, such as an account number. Care must be taken when using keyed files in applications where speed and performance are critical.

☞ *For increased performance, avoid the use of keyed files. After downloads, move frequently used data from CONFIG.SYS to more efficient files.*

File Manager

Omni 300 Series platforms employ a file manager that attempts to recover unused memory space to provide as much data storage space as possible. This space recovery helps ensure maximum transaction storage capacity. Be aware that when writing, inserting, and deleting data,

any files or data stored after the current file pointer are moved. While the time required to move this data is normally imperceptible, the time delay is multiplied when performing multiple operations, such as in a loop. Also, as the application stores additional transaction data, the delay becomes longer as each transaction is added to the same file.

To minimize potential delays resulting from file management routines to reclaim unused memory, observe the following guidelines.

- Place frequently updated files after any large data storage files; this minimizes the amount of data that must be moved during file management.
- Limit the number and frequency of operations that change a file's size.
- When updating records of the same length, overwrite the previous data instead of performing delete and write operations; this eliminates the need for the file manager to move data to occupy the freed space and accommodate the incoming data.

☞ *While fixed length records and padding may be used to prevent changing the size of file records during file management, be aware that these methods will consume valuable transaction storage capacity.*

File Access Functions

With the exception of keyed files, all file access methods use the same set of basic function calls. The method of access is determined by parameters passed to the function. These functions are listed below.

`open()`	Prepares a file for access, allocates and returns a handle for the file
`read()`	Reads data from a file
`write()`	Writes data to a file
`lseek()`	Positions an internal index into the file in preparation for the next `read()` or `write()` operation
`ioctl()`	Returns file-related information, such as its size and the date and time it was last modified
`insert()`	Inserts a record or a specified number of bytes into the file
`delete()`	Deletes a record or a specified number of bytes from the file
`close()`	Frees all internal resources required to access the file

In general, each function returns a specified value greater than or equal to 0 if it is successful, or a value of -1 if it fails. In the latter case, `errno` will be set to a specific error code indicating the nature of the failure.

Opening Files

Before a file can be accessed, it must first be opened. The `open()` function allocates and returns a 2-byte integer file handle which is used in all subsequent file operations.

```
#include <io.h>

handle = open (filename, attributes);
int handle, attributes;
char *filename;
```

The `filename` parameter is a pointer to a NULL-terminated string which may be up to 32 bytes long.

The `attributes` parameter is a 2-byte integer (defined in <fcntl.h>) indicating the type of access attributes to be used:

`O_RDONLY`	Opens the file for read-only access; the file may not be written to
`O_WRONLY`	Opens the file for write-only access; the file may not be read from
`O_RDWR`	Opens the file for read/write access; records may be read, written, inserted or deleted
`O_APPEND`	Opens the file with the file position pointer initially set to the end of the file
`O_CREAT`	Opens a new file for write access
`O_TRUNC`	Opens an existing file, then truncates its length to zero, effectively deleting its previous contents
`O_EXCL`	Used with `O_CREAT` to return an error value if the file already exists.

A single file may be opened multiple times. Each call to `open()` returns a unique handle with access attributes specified by that `open()`. Thus, a file can have multiple

seek pointers in different locations in the file. The programmer is responsible for the consequences of adding or deleting data from a file that has been opened multiple times. The integrity of the file will be maintained, but in some cases it may be difficult to predict where the seek pointers will be positioned.

Up to 30 files may be open simultaneously. Multiple opens of the same file count toward this total. Bear in mind that file handles are a limited resource, and care should be taken in their allocation and use.

Creating New Files

To create a new file, the O_CREAT attribute must be included in the call to open(); O_WRONLY or O_RDWR should be specified to update the newly created file.

If the combination O_CREAT|O_TRUNC is used, an empty file is guaranteed. If you do not wish to write to the file following its opening, omit the O_RDWR or O_WRONLY flag and close it immediately.

If you call open() with O_CREAT and specify an existing filename, O_CREAT is ignored, and the file is simply opened using any additional read/write attributes specified in the call. In this case, a write operation will overwrite any existing data in the file—a potentially destructive situation. To ensure a new file is actually created, include the O_EXCL attribute in the open() call. If the file exists, an error value will be returned to the application, with errno set to EEXIST.

To create a log file to record activity, specify the attributes O_WRONLY|O_APPEND|O_CREAT with the open() call. Log files do not have seek activity during the recording phase.

Opening Files For Writing

Write access to a file is not implied. It must specifically be requested in the open() function by passing O_WRONLY or O_RDWR. O_APPEND can only be used in conjunction with a file that has requested write access.

Files which are opened for O_APPEND have their seek pointers moved to the end of the file. Seeking with `lseek()`, `seek_cvlr()` and `seek_vlr()` can move the file pointer away from the end of a file; however, each write to this file will perform a "seek to the end" before writing data.

File Positioning

Positioning within a file is accomplished using an internal seek pointer—a long integer value—maintained by the file system which contains the byte or record address to be used in the next read, write, insert or delete operation. The seek pointer is allocated when the file is first opened and, therefore, unique per handle. When a file is first opened, the seek pointer is set to a known state, typically zero, the beginning of the file. As noted above, if `O_APPEND` is specified in the `open()` call, the seek pointer will be positioned at the end of the file instead. Applications can modify the seek pointer using the `lseek()`, `seek_vlr()` and `seek_cvlr()` functions.

Reading Data

`read()`
`read_vlr()`
`read_cvlr()`

These functions transfer data from a file that has been opened for reading to a buffer within the application's data area.

```
#include <io.h>

bytes_read = read (handle, buffer, count);
int   bytes_read, handle, count;
char *buffer;
```

A successful call to these functions will copy up to `count` bytes from the file to the address specified by `buffer`. While `count` determines the maximum value to read, the `bytes_read` return value will show the actual number of bytes placed in the buffer; this value may be smaller if the end of file is encountered before the read reaches the `count` value.

`read()` — The file position pointer will point `bytes_read` bytes past its location before the read was executed.

`read_vlr()`, `read_cvlr()` — The file position pointer will point to the next record in the file.

All read statements return `-1` on error, setting the appropriate `errno` value. On error, the seek pointer remains unchanged.

Writing Data

write()
write_vlr()
write_cvlr()

These functions transfer data from an application's buffer to a file that has been opened for writing.

```
#include <io.h>

bytes_written = write (handle, buffer, count);
int  bytes_written, handle, count;
char *buffer;
```

A successful call to these functions will copy up to count bytes from the buffer into the file. The write_vlr() and write_cvlr() functions will either create new records in the file, or overwrite existing records, depending on the position of the seek pointer. While the write_cvlr() function's count parameter determines the maximum value to write, the bytes_written return value will show the actual number of bytes written; this value may be smaller, as write_cvlr() uses compression.

write() — The file position pointer will point to bytes_written bytes past its location before the write is executed.

write_vlr, write_cvlr() — The file position pointer will point to the next record in the file.

All write statements return -1 on error, setting the appropriate errno value. On error, the seek pointer remains unchanged.

If the file was opened with the O_APPEND attribute, all writes are done at the end of the file, regardless of prior calls to lseek(), seek_vlr() or seek_cvlr(). Calling a seek function followed by a read causes the data at that file location to be transferred to the application's buffer. Using O_APPEND means always append.

File Positioning

`lseek()`
`seek_vlr()`
`seek_cvlr()`

These functions set the file position pointer of an open file to a specified location.

```
#include <io.h>

new_position = lseek (handle, offset, origin);
int handle, origin;
long  offset, new_position;
```

To position the seek pointer within a file, pass a starting location and an offset (long integer) value to the function `lseek`, `seek_vlr()` or `seek_cvlr()`, depending on which access method is being used.

Starting locations can be:

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current seek pointer location
`SEEK_END`	End of file

If `SEEK_SET` or `SEEK_END` is used, the system moves the seek pointer to this location and then moves it again, based on the offset value. If `SEEK_CUR` is used, the pointer is moved from its current location by the offset value.

The offset value used with `lseek()` can be positive or negative, and specifies the number of bytes to move the seek pointer from the specified starting point. For the functions `seek_vlr` and `seek_cvlr()`, the offset value must be positive, and it specifies the number of records to seek into the file. Seeking backward in record files is not supported.

The offset value must be specified as a long integer. If the offset value will be passed in the function, remember to terminate the value with an "L" to specify a four-byte value.

Example:

```
bytes=lseek(handle,4L,SEEK_SET);
```

The return value from these functions is the absolute number of bytes (not "records" for seek_vlr and seek_cvlr) from the beginning of the file. In a generic file, this value coincides with the pointer's position in the file. For other types of files, this value is meaningless because it also counts bytes (which includes record headers) instead of records.

Fixed length records may be randomly accessed using the record number as a key. The byte address passed to lseek() is simply the number of records multiplied by the size of each record.

Inserting Data

```
insert()
insert_vlr()
insert_cvlr()
```

These functions insert data into a file opened for write access at the location of the file position pointer:

```
#include <io.h>

bytes_inserted = insert ( handle, buffer,
                          buffer_size );
int   bytes_inserted, handle, buffer_size;
char *buffer;
```

A successful call to these functions will insert up to buffer_size bytes from the buffer into the file. While the insert_cvlr() function's buffer_size parameter determines the maximum value to insert, the bytes_inserted return value will show the actual number of bytes inserted; this value may be smaller because insert_cvlr() uses compression.

The file position pointer is moved to reside at the end of the inserted data.

Deleting Data

```
delete()
delete_vlr()
delete_cvlr()
```

These functions delete data from a file opened for write access at the location of the file position pointer.

```
#include <io.h>

status = delete (handle, count);
int  status, handle, count;
```

Any data following the deleted data is moved to fill the resulting gap. If an error occurs, the returned value of `status` will be -1; if successful, `status` contains no meaningful information.

The file position pointer is not modified by these functions.

Tips on Inserting and Deleting Data

☞ *When adding or deleting data from a file, it is important to remember that any files or data stored in memory after this file will be moved. While normally imperceptible, the amount of time required to perform this move increases with the amount of data being moved.*

☞ *Place frequently updated files after large data-storage files, and limit the number and frequency of operations that change the size of a file. When updating records of the same length, overwrite the previous data rather than deleting the old record and writing the new data. Fixed length records and padding may be used to prevent changing the size of file records during management operations, this should be carefully considered if the overhead will significantly impact transaction storage requirements.*

File Utility Functions

ioctl()

The `ioctl()` function returns information about a specific file: either its size, or when it was last updated. The type of operation performed is based on the constant `GetCtrl`, which is defined in the `<io.h>` header file.

```
#include <io.h>

ioctl (handle, GetCtrl | type, buffer);
int  result, handle;
unsigned  int type;
void  *buffer;
```

The file must be opened before calling `ioctl()`. The type of information returned is determined by `type`, defined as follows:

Type	Description
0x0000	Retrieves a file size as 4- byte long integer returned in `buffer`. File size is determined by size of header plus size of file. The file header consists of: File Size–3 bytes Date/Time–6 bytes Reserved–1 byte Filename size–1 byte (number of characters in the filename) Filename–n bytes (one for each character in the filename) Checksum–2 bytes

❖ *To find the size of the data portion of a binary file, use the call:*

```
data_size = lseek(handle, 0L, SEEK_END);
```

`lseek()` *returns the absolute number of bytes from the beginning of the file and moves the seek pointer to the end of the file.*

Type	Description
0x0001	Returns a 12-byte time stamp in `buffer`. The time stamp contains the date and time the file was last modified in the format: `yymmddhhmmss`, where: `yy`=year, `mm`=month, `dd`=day, `hh`=hour, `mm`=minutes, `ss`=seconds

Closing Files

close()

Each file opened by an application must also be closed when access to the file is no longer needed.

```
#include <io.h>

status = close (handle);
int handle;
```

Once the file has been closed, the `handle` is no longer valid, and all internal resources used by the handle are released.

An alternate way to close files is through the use of `close_all()`:

```
#include <txostd.h>

status = close_all (handle);
int handle;
```

This function releases all active file handles currently in use by the file system. `close_all()` does not close open devices, however. This function should not be called in applications which use TXO buffered file I/O routines such as `fwrite()`. Doing so may cause loss of data.

Example of open(), ioctl(), close()

```
/* FILESYS.C */

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

int file_handle;
long file_size, data_size;
char last_modified[12];

main () {

   /* Open a file. */
   file_handle = open("TEST.DAT", O_CREAT | O_RDWR);

   /* Write 10 bytes. */
   write(file_handle, "0123456789", 10);

   /* Get number of bytes including file header.       */
   ioctl(file_handle, GetCtrl | 0, &file_size);

   /* Get the size of the data portion of the file.    */
   data_size = lseek(file_handle, 0L, SEEK_END);

   /* Display the size of the data portion of the file. */
   printf("SIZE: %ld", data_size);

   /* Get the date and time the file was last modified. */
   ioctl(file_handle, GetCtrl | 1, last_modified);

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

Keyed File Reading and Writing

Keyed files allow records to be accessed by unique character-based strings. In a keyed file, each record consists of two elements: a key value and its associated data.

The same rules for compressed variable length records apply for keyed files. Both key and data values must be text based. Both elements are compressed when stored. Lowercase characters are converted to their uppercase equivalent.

In effect, the mechanism for the keyed file functions (`getkey()` and `putkey()`) are paired, compressed variable length record functions (for key and data).

The maximum length of a key is 32 bytes; data can be up to 128 bytes.

☞ *It is highly recommended that* `getkey()` *and* `putkey()` *functions be used instead of compressed variable length record functions because they are more efficient and avoid file corruption.*

❖ *When ZONTALK 2000 is used to download a keyed file, keys must be 7 bytes or less. ZONTALK 2000 does not restrict data size.*

Keyed files must be created by the system or the application prior to access with `putkey` and `getkey`. If the application creates a file to be used as a keyed file with `open()`, it should immediately `close()` the file to terminate the file handle.

Unlike the other file access methods, keyed files do not need to be opened and closed prior to each read or write. The `getkey()` and `putkey()` functions perform these operations internally.

The `SVC_STORE()` function allows the terminal user to access keyed files by bringing up a "RECALL?" prompt.

Keyed Access Example

```
/* KEYDFILE.C */

# include <stdio.h>
# include <fcntl.h>
# include <strings.h>
# include <errno.h>

/* This program demonstrates keyed access with files */

main ()
  {
     int handle;
     char filename[10];
     char buffer[15];
     char keys[5];
     strcpy (filename,"KEYFILE");


/* Create a file for keyed access and then close it */
     handle = open ("KEYFILE",O_CREAT);
     if (-1 == handle) printf ("\fERROR # %d",errno);
     close (handle);

/* Put "TXO EXAMPLE" into the key "VAR" in the
   file "KEYFILE" and check for error */
     if ( -1 == putkey ("VAR","TXO EXAMPLE",11,filename))
            printf ("\fERROR");

/* Get 14 bytes max of data from key "VAR" in the file
   "KEYFILE" and print it */
     strcpy (keys,"VAR");
     buffer[getkey(keys,buffer,14,"KEYFILE")] = 0;
     printf ("\f%s",buffer);
}
```

getkey()

This function retrieves the data associated with a given key value.

```
#include <txostd.h>

bytes_read = getkey ( key, buffer, max_bytes,
                      file_name );
char *key, *buffer, *file_name;
int max_bytes;
```

The `key` parameter is a null-terminated string up to 32 bytes in length (7 if used with ZONTALK downloading). The `buffer` parameter is a pointer to an array where the data associated with `key` will be stored, with `max_bytes` specifying the size of `buffer`. The `file_name` parameter is a pointer to a null-terminated string which is up to 32 characters long.

The number of bytes read does not necessarily have to match the actual record size. For example, you may read only the first 32 bytes of each record, even though the record may be 120-bytes long. If you want to read the entire record, pass the maximum value of 128 in the `max_bytes` parameter.

If the file does not contain a record matching key, `bytes_read` is returned with a value of zero.

putkey()

The `putkey()` function stores the data for a given key.

```
#include <txostd.h>

result = putkey (key, buffer, count, file_name);
char *key, *buffer, *file_name;
int count, result;
```

The key parameter is a null-terminated string up to 32 bytes in length (7 if used with ZONTALK downloading). The buffer parameter is a pointer to a character array, and count specifies the number of bytes to be written. The file_name parameter is a pointer to a null-terminated string which is up to 32 characters long.

The file being written to must exist; this may be accomplished using open() with the O_CREAT attribute. If putkey() specifies a non-existent file, a -1 result is returned with errno set to EBADF.

The putkey() function may be used to delete a key/record pair by setting the count parameter to zero.

Using Variable Length Records

The variable length record access method lends itself well to storing text-based records of varying lengths, up to a maximum of 254 characters per record. Generic files can be used to store records with varying amounts of data, using the method shown below.

```
#include <io.h>

/* read data of a specified length */
 int   read_data (handle, buff, size);
 char *buff;
 unsigned handle, size;
 {
   read (handle, size, 2);
   read (handle, buff, size);
 }

/* write data of a specified length */
 int   write_data (handle, buff, size);
 char *buff;
 unsigned handle, size;
 {
   write (handle, size, 2);
   write (handle, buff, size);
 }
```

Using these functions, a varying amount of data can be written to or read from a file. When writing data to a file, the data length is first written (assumed to be a 2-byte integer), followed by the data itself.

This method is particularly useful when file data is processed serially; it does not lend itself well to random access.

The preceding examples only demonstrate the concept of using a two-step read or write operation. To use this technique in your application, some additional logic is needed. For example, if the first read in `read_data()` failed, an early exit would be desirable, perhaps returning the contents of the errno variable to the calling routine.

Variable Length Records Example

```
/* VLRFILE.C */

/*  File demo for variable length data records. This program shows
    correct syntax for using file functions. The program executes
    all statements and then terminates. */

# include <stdio.h>
# include <io.h>
# include <fcntl.h>
# include <strings.h>
# include <memory.h>
# include <vlr.h>
# include <errno.h>
# include <txosvc.h>

char buffer [13];

main ()
   {
     int handle, bytes_written, bytes_read;
     int bytes_inserted, del_chk, result;
     long int total_size, position;
     char i;
     static char file_name[] = "VLRFILE.TXT";
```

```
/* Create (if it does not exist) the file VLRFILE.TXT for
   read and write - file pointer is set to SEEK_SET */
     handle = open (file_name,O_CREAT | O_RDWR);
     if (-1 == handle) printf ("\fERROR #  %d",errno);

/* Write "1233456789" to VLRFILE.TXT - file pointer will
   be at end of file */
     strcpy (buffer,"123456789");
     bytes_written = write_vlr(handle,buffer,strlen(buffer));
     if (-1 == bytes_written) printf ("\fERROR # %d",errno);

/* Write "ABCDEFGHIJK" as second record in VLRFILE.TXT - file
   pointer will be at end of file */
     strcpy (buffer,"ABCDEFGHIJK");
     bytes_written = write_vlr(handle,buffer,strlen(buffer));
     if (-1 == bytes_written) printf ("\fERROR # %d",errno);

/* Set file pointer to first record in file */
     position = seek_vlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);

/* Fill buffer with nulls then read 6 bytes max from the
   current record - "123456" will be put in buffer and
   the file pointer will move to the next record */
     memset (buffer,0,13);
     bytes_read = read_vlr(handle,buffer,6);
     if  (-1 == bytes_read) printf ("\fERROR # %d",errno);

/* Fill buffer with nulls then read 12 bytes max from the
   current record - "ABCDEFGHIJK" will be put in buffer and
   the file pointer at the end of the file */
     memset (buffer,0,13);
     bytes_read = read_vlr(handle,buffer,12);
     if  (-1 == bytes_read) printf ("\fERROR # %d",errno);

/* Set file pointer to first record in file */
     position = seek_vlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);

/* Replace the first record with "TESTING 123" - file pointer will
   then be at second record */
     strcpy (buffer,"TESTING 123");
     bytes_written = write_vlr(handle,buffer,strlen(buffer));
     if (-1 == bytes_written) printf ("\fERROR # %d",errno);
```

```
/* Insert "HELLO WORLD" before second record. Second record
   is moved to third and file pointer is at third record. */
     bytes_inserted = insert_vlr (handle,"HELLO WORLD",11);
     if (-1 == bytes_inserted) printf ("\fERROR # %d",errno);

/* Set file pointer to first record in file and print all
   three records */
     position = seek_vlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);
     result = 1;
     while (0 != result)
          {
               memset (buffer,0,13);
               result = read_vlr(handle,buffer,12);
               printf ("\f%s",buffer);
               SVC_WAIT(1000);   /* WAIT FOR 1000 MS */
          }

/* Set file pointer to the first record and delete both the first
   and second record. The only record left will be "ABCDEFGHIJK"
   which will become the first record. */
     position = seek_vlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);
     del_chk = delete_vlr (handle,2);
     if (-1 == del_chk) printf ("\fERROR # %d",errno);

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

Using Compressed Variable Length Records

The file system provides the capability to read and write variable length records containing compressed data, referred to as compressed variable length records (CVLR). When the data is retrieved it is decompressed.

❖ *The programmer must ensure that records written as variable length records are read as variable length records, and that the records written as compressed records are read as compressed records.*

Compressed variable length records are arrays of up to 254 bytes stored with a count-byte prefix. The count byte—designed by the operating system—allows the file system to maintain data as separate records; the count byte requires one byte of additional space for each record in the file.

Data compression operates on a subset of the ASCII character set. The algorithm uses four bits to store numeric characters ("0"–"9") and eight bits to store all other characters. More specifically, in order to recognize a non-numeric character (that is, to detect that the following eight bits make up a single character), the algorithm forces all non-numeric characters to be encoded into an eight-bit string beginning with hex digits 'A' through 'F,' allowing for 96 (6*16=96) unique combinations. This means that the compression algorithm is not completely reversible. For example, each lowercase character is replaced by its uppercase equivalent.

In Table 6-1 on the opposite page, each possible byte value (matching the is represented after it is compressed.
For example:

	Compressed	
Hex	Hex	ASCII
30	0	0
41	E1	A
61	E1	a

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
1-	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
2-	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
3-	0	1	2	3	4	5	6	7	8	9	DA	DB	DC	DD	DE	DF
4-	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
5-	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF
6-	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
7-	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF
8-	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
9-	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
A-	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
B-	0	1	2	3	4	5	6	7	8	9	DA	DB	DC	DD	DE	DF
C-	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
D-	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF
E-	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
F-	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

Table 6-1. Compressed Character Storage

Note that all input characters above 0x5F will not be translated correctly—that is, when retrieved from storage, a different character will be returned.

Compressed Variable Length Records Example

```
/* CVLRFILE.C */

/*  File demo for compressed variable length data records. This
    program shows correct syntax for using file functions. The
    program executes all statements and then terminates. */

# include <stdio.h>
# include <io.h>
# include <fcntl.h>
# include <strings.h>
# include <memory.h>
# include <vlr.h>
# include <errno.h>
# include <txosvc.h>

char buffer [13];

main ()
   {
     int handle, bytes_written, bytes_read;
     int bytes_inserted, del_chk, result;
     long int total_size, position;
     char i;
     static char file_name[] = "CVLRFILE.TXT";

/* Open (create if it does not exist) the file CVLRFILE.TXT for
   read and write - file pointer is set to SEEK_SET */
     handle = open (file_name,O_CREAT | O_RDWR);
     if (-1 == handle) printf ("\fERROR #  %d",errno);

/* Write "1233456789" to CVLRFILE.TXT - file pointer will
   be at end of file */
     strcpy (buffer,"123456789");
     bytes_written = write_cvlr(handle,buffer,strlen(buffer));
     if (-1 == bytes_written) printf ("\fERROR # %d",errno);

/* Write "ABCDEFGHIJK" as second record in CVLRFILE.TXT - file
   pointer will be at end of file */
     strcpy (buffer,"ABCDEFGHIJK");
     bytes_written = write_cvlr(handle,buffer,strlen(buffer));
     if (-1 == bytes_written) printf ("\fERROR # %d",errno);
```

```
/* Set file pointer to first record in file */
     position = seek_cvlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);

/* Fill buffer with nulls then read 3 bytes max from the
   current record - "123456" will be put in buffer and
   the file pointer will move to the next record */
     memset (buffer,0,13);
     bytes_read = read_cvlr(handle,buffer,3);
     if  (-1 == bytes_read) printf ("\fERROR # %d",errno);

/* Fill buffer with nulls then read 12 bytes max from the
   current record - "ABCDEFGHIJK" will be put in buffer and
   the file pointer at the end of the file */
     memset (buffer,0,13);
     bytes_read = read_cvlr(handle,buffer,12);
     if  (-1 == bytes_read) printf ("\fERROR # %d",errno);

/* SEt file pointer to first record in file */
     position = seek_cvlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);

/* Replace the first record with "TESTING 123" - file pointer will
   then be at second record */
     strcpy (buffer,"TESTING 123");
     bytes_written = write_cvlr(handle,buffer,strlen(buffer));
     if (-1 == bytes_written) printf ("\fERROR # %d",errno);

/* Insert "HELLO WORLD" before second record. Second record
   is moved to third and file pointer is at third record */
     bytes_inserted = insert_cvlr (handle,"HELLO WORLD",11);
     if (-1 == bytes_inserted) printf ("\fERROR # %d",errno);

/* Set file pointer to first record in file and print all
   three records */
     position = seek_cvlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);
     result = 1;
     while (0 != result)
          {
               memset (buffer,0,13);
               result = read_cvlr(handle,buffer,12);
               printf ("\f%s",buffer);
               SVC_WAIT(1000);   /* WAIT FOR 1000 MS */
          }
```

```
/* Set file pointer to the first record and delete both the first
   and second record. The only record left will be "ABCDEFGHIJK"
   which will become the first record. */
     position = seek_cvlr (handle,0L,SEEK_SET);
     if  (-1 == position) printf ("\fERROR # %d",errno);
     del_chk = delete_cvlr (handle,2);
     if (-1 == del_chk) printf ("\fERROR # %d",errno);

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

File Directory Functions

The file system uses a non-hierarchical directory, i.e., no subidrectories. File names may be up to 32 characters in length and must be terminated by a NULL.

dir_get_sizes()

This function returns general information about the directory: the number of files in the directory, the amount of memory used by the file system, and the amount of free space remaining.

```
#include <txostd.h>

dir_get_sizes (buffer);
char *buffer;

/* returned values in buffer: */
int  filecount;
long  currentsize, availablesize;
```

Upon return, buffer is filled with 10 bytes of information about the file system. The value of filecount indicates the number of files in the directory. The currentsize field contains the amount of memory used by the file system, including memory manager overhead (in bytes). The availablesize field contains the amount of remaining space in the file system.

dir_get_first()

This function returns a null-terminated string containing the name of the first file in the directory (usually CONFIG.SYS).

```
#include <txostd.h>

result = dir_get_first (buffer);
char *buffer;
int  result;
```

Upon return, `buffer` contains the null-terminated name of the first file found in the directory, with `result` set to zero if the call was successful. A non-zero value returned in `result` indicates that the directory has been damaged.

dir_get_next()

This function is normally called after `dir_get_first()` and is used to get subsequent file names from the directory.

```
#include <txostd.h>

result = dir_get_next (buffer);
char *buffer;
int  result;
```

The `buffer` parameter normally contains the name of the file returned from a prior call to `dir_get_first()`.

The directory is searched for the specified file name, and the name of the file in the following entry is returned. If the file name passed in `buffer` is not found or is the last entry in the directory, `result` is set to -1, with `errno` set to `ENOENT`.

remove()

❖ *Prior to calling this function, all open handles for the target file must be closed. This includes any multiple opens of the target file.*

This function removes (deletes) a specified file in the directory.

```
#include <txostd.h>

result = remove (buffer);
int  result;
char *buffer;
```

The name of the file to be deleted is placed in `buffer` as a NULL-terminated string. If the specified file is found, the file is deleted and `result` is set to zero. If the file is not found, `result` is set to -1 and `errno` is set to `ENOENT`.

Directory Function Example

```
/* DIR.C */

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


/* This program demonstrates directory functions. Remember that
   the system has created four files: CONFIG.SYS, CODE.EM, DATA.EM,
   and SYSTEM.BIN. Three new files are opened but then one of them
   is removed. A total of six files will remain in the directory.
*/


main ()
   {
     int handle[3];
     char dir_buff [12];

/* Define structure/union to hold directory info */

union {
      struct s_dir {
         int filecount;          /* Number of files in directory */
         long currentsize;       /* Number of bytes in use       */
         long availablesize;     /* Number of bytes remaining    */
                  }  directory;
      char dir_line[10];
      } dline;

/* Create three files */
     handle[0] = open ("FILE_ONE",O_CREAT);
     if (handle[0] == -1) printf ("\fERROR # %d",errno);
     handle[1] = open ("FILE_TWO",O_CREAT);
     if (handle[1] == -1) printf ("\fERROR # %d",errno);
     handle[2] = open ("FILE_THREE",O_CREAT);
```

```
      if (handle[2] == -1) printf ("\fERROR # %d",errno);

/* Close all files and then remove "FILE_TWO" from directory */
      close_all();
      if (-1 == remove("File_tWO")) printf ("\fERROR");

/* Read directory information  */
      dir_get_sizes (dline.dir_line);

/* Print directory information */
      printf ("\f%d FILES",dline.directory.filecount);
      SVC_WAIT(1000);
      printf ("\f%ld BYTES USED",dline.directory.currentsize);
      SVC_WAIT(1000);
      printf ("\f%ld BYTES FREE",dline.directory.availablesize);
      SVC_WAIT(1000);

/* Get the ID of the first file in the directory. A value of zero
   is returned upon success  */
      memset (dir_buff,0,12);
      if(0 != dir_get_first(dir_buff))
            printf ("\fSEVERE ERROR");
      printf("\f%s",dir_buff);
      SVC_WAIT(1500);

/* Get the ID of the next file in the directory after the file
   contained in dir_buff until end of directory. The buffer
   dir_buff must contain a valid file ID. */

      while (-1 != dir_get_next(dir_buff))
            {
            printf("\f%s",dir_buff);
            SVC_WAIT (1500);
            }
      printf ("\fEND EXAMPLE");
```

C H A P T E R

7

TXO Exception Handling

Unique to TXO firmware is an event/exception handling system that allows the application to immediately process terminal events or exceptions using a series of *traps* and *guards*.

A *trap* "flags" the occurrence of an event or exception. A *guard* then handles this trapped event/exception by calling an application-defined function to process the event/exception that has occurred, and manage other events/exceptions that may occur during the exception handling process.

Traps

The exception-handling system recognizes 64 traps: 60 are system-defined, while four may be defined by the user. Each trap is associated with a specific trap number defined in the `<trap.h>` include file. Traps are numbered 0-63, as described in the table on the following page.

Trap Numbers

The following table lists traps defined and available for OMNI 300 Series terminals. All unlisted traps are reserved and should not be used.

Operating System Traps		
0 –15		Operating system advisory traps (except floating point underflow).
Severe Error Traps		
16 – 27		The various conditions that may cause one of these traps to occur often signal that the application has become corrupt, and is no longer reliable. Rather than continue the application, and risk some other failure, the application should set a flag in the system CONFIG.SYS file to show that a severe error has occurred, call `SVC_RESTART()` to re-boot the application, and display a message to the terminal user, such as "`CALL HELPDESK`".
❖ *If the application has not armed one of these traps, and a serious error occurs, the operating system intervenes and the terminal displays:* `EXEC ERROR NN`, *where* `NN` *is the trap number.*		
User-defined Traps		
28	User Trap #4	Traps 28–31 are provided for use by application-specific functions and can only be set by an application call. If one or more of these traps are used, it is the application's responsibility to arm, dispatch and disarm them.
29	User Trap #3	
30	User Trap #2	
31	User Trap #1	
❖ *Traps 0-31 are the same on 400 Series terminals. However, for improved portability, use the* `trap_of()` *routine to get a trap number.*		
Device Traps		
32	Keyboard	[CLEAR] key pressed. See *Clear Key Processing*, page 7-12.
33	Keyboard	Any key other than [CLEAR] key.
34	Keyboard	Invalid System Password, or the system times out while the "SYSTEM PASSWORD?" prompt appears. (see *PTO environment variable in CONFIG.SYS file).
38	Clock	Signaled approximately every 1 second.
39	Clock	System tick signaled. The system tick occurs 64 times every second on OMNI 300 Series terminals. Because of the high frequency of this trap, it not recommended for use except in very extreme circumstances.

Device Traps (continued)		
40	Modem	Signaled when system detects carrier loss.
41	Modem	Signaled when the modem responds to a Hayes command. Each response is terminated by a carriage return character, and is available on the `read_cmd()` line of the modem. This trap also signals when carrier is lost, as loss of carrier is a valid Hayes response (3\r).
48	Modem	Output Failure. See *Device Notes.*
49	Modem	Input Available. See *Device Notes.*
42	COM1 Port	Output Failure. See *Device Notes.*
43	COM1 Port	Input Available. See *Device Notes.*
46	COM3 Port	Output Failure. See *Device Notes.*
47	COM3 Port	Input Available (on select 300 Series Terminals). See *Device Notes.*
52	LAN Port	One or more LAN exception(s)/event(s) has occurred.
53	LAN Port	LAN data packet received.
57	Card Reader	Input Available. Signaled on a card swipe. In order to set this trap, the card device must be opened and the operating system card swipe buffer must be empty.
59	Bar Code	Input Available
60	PIN Pad	Output Failure. See *Device Notes.*
61	PIN Pad	Input Available. See *Device Notes.*

Device Notes

- *Most devices are assigned two trap numbers. One signals completion of normal input, while the other trap signals an error. The bar code and card reader devices only signal normal input completion.*
- *Output Failures: The transmission output failure trap is provided only for intelligent protocols. On the modem, only the VISA First Generation Protocol is currently supported. When a packet is transmitted but not ACKed by the host, the system will set this trap. Character mode and packet mode are not intelligent protocols and do not use this trap. COM1, COM3 or the PIN Pad ports do not provide traps for intelligent protocols.*
- *Input Available: Signaled when data input completes (e.g., data is available on the `read()` line of the modem). In character mode, every byte is treated as an event. In packet mode, complete packets are treated as one event, and any bytes outside of the packet framing are treated as separate events.*

Guards

The exception-handling system uses guards to monitor events and call an appropriate application-defined *guard function* to process the event or exception when it occurs. This system allows up to 16 guards to be *armed* (set to monitor exceptions/events) at any given time. Of these 16 guards, some may be set up to handle user-defined exceptions such as program errors, as illustrated in the example program file USERTRAP.C.

Guards are set to monitor events/exceptions by the library function `arm_guard()`, which is made up of four elements: a *take mask*, a *hide mask*, a *guard function*, and a *slot number*. The following is an illustration of an `arm_guard()` call taken from the example file GUARD.C:

```
slot4 = arm_guard (other_key, mask1, hide_all);
```

Take Mask The take mask identifies the event(s) (trap numbers) the guard will monitor. Each guard can monitor from 1 to 64 traps. If any events identified by the `arm_guard()` function occur, the guard invokes its associated guard function.

Hide Mask The hide mask identifies events to be ignored while its associated guard function is running. Typically, this mask is identical to the take mask. To avoid any interruptions while a guard function is executing, use a hide mask produced by the `mask_fill()` library routine.

Guard Function A guard function is a user-defined routine that handles the identified events or exceptions. Typically guard functions manage system I/O events from the keyboard, cardreader or modem, or clock events. Guard functions can also handle program errors when used with user-defined traps. Instructions for writing these functions are provided later in this section.

Slot Number The slot number returned by `arm_guard()` is placed in an internal *guard list*. When an identified event or exception occurs, its bit in the system trap mask is set, and the system compares its internal trap mask to each

guard's take mask to determine which guard is monitoring the event that has occurred. The system examines each guard in the guard list—from the most recently set guard (youngest) to the oldest. The first guard found to be monitoring the event that has occurred calls its associated guard function to handle the event.

As shown in the guard list management example below, if multiple guards are monitoring the same event, only the youngest (most recently set) guard is called.

❖ *If other guard functions are running, the trap may not be reset immediately. See Hiding Events later in this section.*

Guard List Management Example

In the following example, the guard in Slot 3 was the last guard armed, and is therefore the youngest guard. If the [CLEAR] key is pressed, only the guard function `keys_f()` is called. The function `clear_key()` will not be called in this example until the guard in Slot 3 is disarmed.

❖ *The slot number returned from the* `arm_guard()` *call is used only as a place holder in the system—it gives no indication of the order in which guards were armed (youngest or oldest). The seniority of the guards is based on the order in which they were activated by* `arm_guard()`.

It is the task of the application to monitor guard senority.

Example Guard List

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

Pending Events

Events are only *pended* when a guard function is running and that guard function has identified "events to ignore" in its hide mask. In the above example, the function `keys_f()` (monitoring both Trap 33 and Trap 32) is called immediately