Chaining Example

In the example below, assume the following three programs are built and downloaded using the command:

```
DL CHAIN1.OUT CHAIN2.OUT CHAIN3.OUT *GO=""
```

Starting the terminal's application will result in the following display:

```
RUN CHAIN1.OUT
RUN CHAIN2.OUT
RUN CHAIN3.OUT
RUN CHAIN1.OUT
...
```

The source code files read as follows:

Assume the following three programs are built and downloaded using:
DL CHAIN1.OUT CHAIN2.OUT CHAIN3.OUT *GO=""

```
/* CHAIN1.C */

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

main(argc,argv)    int argc; char *argv[];  {

   /* Display the name of the currently executing application. */
   printf("\fRUN %s", argv[argc-1]);

   /* Start CHAIN2.OUT */
   SVC_RESTART("CHAIN2.OUT");
}
```

```
/* CHAIN2.C */

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

main(argc,argv)    int argc; char *argv[];  {

   /* Display the name of the currently executing application. */
   printf("\fRUN %s", argv[argc-1]);

   /* Set *GO=CHAIN3.OUT */
   put_env("*GO", "CHAIN3.OUT", 10);

   /* System will restart application that *GO is set to. */
   SVC_RESTART("");
}

/* CHAIN3.C */

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

main(argc,argv)    int argc; char *argv[];  {

   /* Display the name of the currently executing application. */
   printf("\fRUN %s", argv[argc-1]);

   /* Delete the *GO entry from CONFIG.SYS */
   put_env("*GO", "", 0);

   /* Restart the oldest code file (CHAIN1.OUT). */
   SVC_RESTART("");
}
```

Starting the terminal's application will result in the following display:

```
RUN CHAIN1.OUT
RUN CHAIN2.OUT
RUN CHAIN3.OUT
RUN CHAIN1.OUT
```

TXO Programming Tips and Guidelines

The following tips are based on the collective experience of many VeriFone programmers, analysts and engineers.

☞ *Use global variables only when absolutely necessary. In most cases, a value or pointer to a value can be passed from one function to another without resorting to the use of global variables. This is of particular benefit in program maintenance and troubleshooting. You may temporarily change local variables to global variables to watch their values change with the symbolic debugger, and then restore them to local variables after the application is debugged.*

☞ *If a single application program uses more than 45Kb of program space, use one of the proposed methods for handling large applications—either R-modules or chained files. See Handling Large Applications discussed earlier in this section. Using multiple executable code modules for an application creates additional global space (each module may have 14Kb of global and stack space), allows larger programs (each module may be 64Kb), facilitates troubleshooting (each module may be individually developed and tested), decreases maintenance downloads (an individual module can be downloaded), and allows general purpose code modules to be developed and reused in other applications.*

☞ *Design file management routines to minimize the need to increase or decrease file size. Reusing record space when possible instead of deleting and inserting new records can significantly improve the application's performance.*

☞ *For increased performance, avoid the use of keyed files. After downloads, move frequently used data from CONFIG.SYS to a data file that is not a keyed file.*

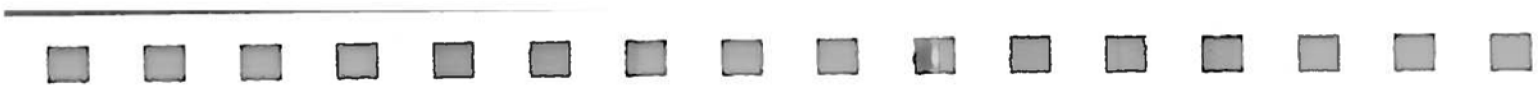
☞ *Design your application to rely on downloaded data files and table-driven techniques to make it more flexible, and easier to maintain and update.*

☞ *Data entry response time can be improved by taking full advantage of the two-line display (if available). Use one line for prompts and the other for data entry. Use data entry routines that support keyboard buffering and terminate display delays as soon as data entry begins.*

☞ *Take advantage of operating system buffers to avoid processing delays associated with external devices. Several lines of data can be sent to a device with a single write operation.*

☞ *Avoid processing initialization files when no changes have occurred. Use the file date/time stamp to determine if initialization is needed.*

☞ *To decrease transaction processing time, use predialing (in dial-type models) and preprinting techniques. Connecting to a host can take a considerable amount of time. Early dialing allows the application to complete some of the* data entry while the host connection is being established. *Header information can be printed during data entry, or can be performed as a post-transaction process. At the end of every transaction, the header information for the next receipt can be printed. The only disadvantage is that paper will be wasted if the transaction is not completed or denied.*



C H A P T E R

4

System Configuration File

TXO terminals use a default system file, CONFIG.SYS, to set up the system environment. An end user can add or change CONFIG.SYS entries using the System Mode file editor. An application may read and write to the file through the `get_env()` and `put_env()` library routines.

CONFIG.SYS is a compressed ASCII format file maintained as a keyed file (see *Keyed Variable Length Records* in Chapter 6). Entries are stored in pairs of variable length records as:

key, data

The terminal displays these entries as:

key=data

To preserve entries during a full LAN or PC download, prefix the key with a pound sign (#).

❖ *CONFIG.SYS entries beginning with an asterisk (*) are also preserved during full downloads. However, this character denotes a system variable and should not be created by an application or the terminal user.*

- *CONFIG.SYS entries should not use control codes (values between 0x00 and 0x1F), as they are reserved for future use. Improper use of control codes in the file may cause malfunctions during download operations.*

- *In terminal-to-terminal downloads, all data in the receiving terminal—including CONFIG.SYS records beginning with an asterisk or the pound sign—is erased and replaced with an image of the sending terminal's memory.*

Environment Variables

Below are the currently used CONFIG.SYS environment variables and their descriptions:

Entry	Description
*B	Sets the number of communication buffers. Default and minimum of 4, maximum of 32.
CHKSUM	Disables checksum verification on startup. CHKSUM=1 checksum enabled, CHKSUM=2 checksum disabled.
*CHN	Flag to speed up application restarts (DATA.EM file not deleted and reconstructed). Not available in EPROM versions 15 or greater. *CHN=1 enables speeds up application restart.
*CL	Bilingual Support.
*D	Sets port and speed for application debugging. Data format of *D=ppbb (diagnostics port number and baud rate).
*FONT	Sets default font for pixel display
*GRID	Sets default grid for pixel display
*LAD	LAN terminal address. Must be in range 1-32.
*LBR	LAN baud rate. 4=4800, 5=9600, 6=19,200.
*LDS	LAN download server. Must be in range 1-32, or 254.

Entry	Description
*LFP	LAN full or partial download
*LHA	LAN high terminal address. Must be in range 1-32.
*LTW	LAN timing window. 1=fast, 2=slow.
*LZF	LAN download all terminals flag. Value="A".
*MI	Modem initialization
*PTO	Password timeout. Must be in range 1-999 seconds.
*S	Increases stack size
*T	Remote diagnostics host phone number
*ZA	ZONTALK application ID
*ZP	ZONTALK host telephone number. May use imbedded dialing control characters.
*ZR	ZONTALK rate. 0=300, 2=1200, 3=2400, ... Default is 1200.
*ZT	ZONTALK terminal ID
*ZX	Automatic application startup after download when *ZX=1

*B – Communication Device Buffers

The system maintains a set of memory buffers for communication device I/O operations (RS-232, PIN Pad, modem, LAN, etc.). All I/O operations that you are using simultaneously share the communication buffer pool. Increasing the number of communication buffers improves I/O function performance, but it also increases memory use. The programmer must therefore decide which is more important to the application. *B accepts a decimal number indicating the number of communication buffers the system maintains. For example, *B=24 assigns 24 buffers. Each buffer comprises 254 bytes.

The default value is 4, the minimum number of buffers the system allows. If a value of less than 4 is entered,

the system still defaults to 4 buffers. The maximum number of buffers is 32.

☞ *Although 4 buffers may be adequate for testing, it may not be enough for real-time application.*

☞ *A lack of buffers may cause I/O problems, such as the failure of* `write()` *commands or LAN operations. Therefore, it is good practice to check return codes from I/O operations of the communications devices. The* `write()` *command returns 0 bytes written if write fails due to exhausted buffers.*

In VISA mode, if packets are rejected, buffers are used to queue the rejected packets.

On reads, the buffers can overflow if reads do not keep up with incoming data. On writes, the buffers can be exhausted if more data is written than there are available buffers. The maximum reads or writes is 4064 bytes with *B set at the maximum.

Non-communications devices (keyboard, card reader, etc.) do not use the buffers allocated via *B and are not affected by this variable.

CHKSUM – Checksum Control

By default, each time the system "starts" the application, it validates the checksums of all the files in the file system (on power-up, or following an exit from system mode). The user may reduce or bypass automatic checksum verification by assigning a value of 2 to CHKSUM.

If CHKSUM=1, the system checks all its files when the application is started as a result of a power-up, but does not check on exit from system mode.

If CHKSUM=2, the system bypasses checksum validation at all times.

Any other CHKSUM value is undefined.

If automatic checksum validation is disabled, the application can check the integrity of its files using the `SVC_CHECKFILE()` function.

- *CHKSUM does not begin with an asterisk and will be deleted from CONFIG.SYS on a full download. This ensures that checksum validation is not disabled when a new application is loaded into the system.*

*CHN – Speed Up Application Restarts

- *EPROM versions 15 or greater do not use this variable. The application data file is zeroed out upon application startup.*

During application startup, the system normally deletes the existing application data file (DATA.EM—which holds globals, locals, stack, etc.) and creates the file anew. This operation can take precious time, particularly when the application consists of multiple files chained together via `SVC_RESTART()`.

To reduce startup time associated with the deletion and re-creation of the data file, you may set *CHN=1. With this setting, the data file is not deleted on system restart, but retains its current size (or expands, if necessary). *The user's application data is always re-initialized during startup.*

*CL – Bilingual Support

Certain international versions (e.g., Canada) can now be built with limited support for two languages. The operator will be asked to select which set should be used for (nearly all) system messages. This selection is stored in CONFIG.SYS as the *CL entry.

- *Bilingual support is provided in EPROM Versions 20 and above.*

Entry	Action
*CL = 1	Lang. #1 (corresponding to patch table #1)
*CL = 2	Lang. #2 (corresponding to patch table #2)

Proper setting of this CONFIG.SYS variable will set up the terminal with all display prompts in the selected language, with exception of:

```
MEMORY ERROR
PROGRAM ERROR
FILE ERROR
BAD EPROM
'* MEMORY TEST *'
* MEMORY -OK- *'
*** BAD RAM ***'1=ENGLISH 2=FRNCH'
```

The above prompts may be changed via PATCH provided the lengths of each string are preserved exactly.

*D – Debugging Control

This entry specifies the communication settings for the TXO symbolic debugger. Enter data in the form:

*D=*ppbb*

where:

pp Diagnostics port number in hexadecimal:
- 0A = COM1 port
- 1C = PIN Pad/Bar Code port

bb Baud rate:

00 = 300	04 = 4800
01 = 600	05 = 9600
02 = 1200	06 = 19200
03 = 2400	

The normal setting is *D=0A06 (COM1, 19200 baud rate).

*FONT, *GRID – Changing Initial Font/Grid *(pixel-type displays)*

❖ *These variables are only used by terminals equipped with a bit-mapped, pixel-type display. They have no effect on segment-type displays.*

A terminal's display font and grid may be changed at application startup by setting the CONFIG.SYS entries *FONT and *GRID to the desired Font Definition File and

the desired grid, respectively. See *Chapter 6, System Devices, Default Font Definition File.* For either entry to take effect, both must be set. If either setting is invalid, the default settings will remain. These settings do not affect the display during system mode operation.

*GO – Startup Executable Code File

Upon powerup or a system restart, the terminal will decide which program to run by looking at the *GO entry in CONFIG.SYS.

For example, if *GO = APPL.OUT, upon system restart the terminal searchs for the file APPL.OUT and attempts to execute it. If the file is not found the terminal will display "INVALID *GO PARM" and any keypress will go to System Mode. If the file is not a valid EM code file, the terminal will display "PROGRAM ERROR" and any keypress will go to System Mode.

If *GO is not set at system restart, the terminal looks for the first EM code file and attempts to execute it. If no files are found, or there are no valid EM applications, the terminal will display "DOWNLOAD NEEDED" and any keypress will go to System Mode.

Executables are not required to have an ".OUT" suffix. Also, the filename of the currently executing program can be determined from `argv` argument (in `main()`). `argc` and `argv` are currently supported in a very limited manner. `argc` is always 1 and `argv[0]` is the filename of the currently executing program.

*L Series – LAN Control

The following entries may be used in the CONFIG.SYS file to control Local Area Network (LAN) functions for OMNI 300 Series LAN terminals:

*LAD	LAN Terminal Address
*LBR	LAN Baud Rate
*LDS	LAN Download Server
*LFP	LAN Full or Partial Download
*LHA	LAN High Address
*LTW	LAN Timing Window
*LZF	LAN Download All Terminals Flag

*LAD - LAN Terminal Address

Each terminal on the LAN must have a unique terminal address or communication problems may occur. This address must be in the range of 1 to 32—the maximum value of *LHA (LAN high address) is 32. There is no default for this CONFIG.SYS variable, and it must be set.

*LBR - LAN Baud Rate

This parameter is required for both LAN downloads and normal LAN communications. It specifies the rate at which downloads and communications will take place on the LAN. The value of this parameter must be the same on all LAN terminals or the LAN will not function correctly.

Valid settings for this parameter are: 4 (4800 baud), 5 (9600 baud), or 6 (19,200 baud). This parameter must be set before any LAN downloads are performed; there is no default value. If downloads are accomplished in some other manner, the LAN can be opened by the application (using the `put_lan_config()` function) without explicitly setting this parameter, and the default value is set to 6. This value is also the recommended value for most LANs since it will provide for the fastest data throughput.

*LDS - LAN Download Server

This parameter sets the address of the download server; a value of 1 to 32, or 254 may be entered. When this parameter is set to 254, a broadcast download will be

performed to *all* terminals on the LAN whenever a LAN download is requested. This parameter will be prompted for (if not previously set) during the initiation of the LAN download (by pressing [0] from System Mode).

This parameter has no default value, and it is not required if downloads are not to be performed via the LAN.

*LFP - LAN Full or Partial Download

This parameter will be prompted for (if not previously set) during the initiation of the LAN download (by pressing [0] from System Mode). The user is shown the prompts `PARTIAL OR FULL?` and `"PART=• FULL=FUNC.`

A partial application download adds files to what is already in the terminal. A full application download replaces all programs and data in the terminal, except for protected records in the terminal's CONFIG.SYS file (records with keys that start with either the "*" or "#" characters).

This parameter has no default value, and it is not required if downloads are not to be performed via the LAN.

*LHA - LAN High Address

This parameter is required for both LAN downloads and normal LAN communications. It specifies the highest terminal address expected to exist on the LAN. The value of this parameter *must* be the same on all terminals on the LAN or the LAN will not function correctly. It is used by the operating system to calculate the "slots" or time intervals in which each terminal may transmit across the LAN, and terminals calculating these slots differently will create "collisions" on the LAN.

When setting this parameter, be aware that if the high address is set to number of terminals on the LAN, and more terminals are later added, *all* LAN terminals will need to be reconfigured. However, the higher the high LAN address value, the slower the overall throughput. If you are certain that the number of terminals on the LAN will not change, set the high address to that

number, otherwise ancitipate LAN expansion and set the high address accordingly.

The default high address is 32 if the application attempts to open the LAN with the `put_lan_config()` function. There is no default for uninitialized terminals; the user will be prompted to enter the high address.

*LTW - LAN Timing Window

This parameter selects the desired rate of data throughput. It may either be set at 1 (fast), which is the normal setting, or at 2 (slow), which is preferred for LANs that experience electrical interference. The "fast" setting is preferable since it allows faster LAN throughput.

This parameter is required for both LAN downloads and normal LAN communications. It specifies the "timing window" on the LAN. The value of this parameter *must* be the same on all terminals on the LAN or the LAN will not function correctly.

The default is 1 if the application attempts to open the LAN with the `put_lan_config()` function. There is no default for uninitialized terminals; the user will be prompted to set the timing window.

*LZF - LAN Download All Terminals Flag

This is an optional LAN parameter. If present, it must have the value "A" (for all) and will cause the download request message issued by *any* terminal on the LAN to be interpreted as a request to download to *all* terminals on the LAN.

*MI – Modem Control

The *MI variable is used to change modem default settings on dial-type terminals using Hayes-compatible controls.

See *Chapter 9, Communications Devices, Modem*, for details.

*PTO – Password Timeout

*PTO specifies the maximum number of seconds—between 1 and 999—allowed between password keystrokes. If the timeout period lapses between any of the keystrokes required to complete the password, the password request is canceled and the application resumes execution. If *PTO is not set, or is set to an invalid value, the terminal will not timeout during password entry.

*S – Increasing Stack Size

A system file generated by the TXO Workbench defines the minimum stack size required to run the program. This allocated space may be increased by setting *S to a decimal number value representing the additional allocation size in bytes. For example, if the default stack size is 1000 bytes and *S=3000 is entered, the new stack size is 4000 bytes.

During a system restart, the message "STACK TOO BIG" indicates that the application is using too much stack space. If this happens, the terminal waits for a keystroke and then goes into System Mode. The stack size must be reduced either by decreasing the *S entry or by adjusting and recompiling the application so that it uses less stack space and downloading it again.

*T — Remote Diagnostics Host Telephone Number

This variable (*T = *xxxxxxx*) contains the telephone number of a remote diagnostics computer. See your terminal's *Reference Manual* for remote diagnostics operation. This parameter only applies to dial-type terminals.

*Z Series — ZONTALK 2000 Control

The following five entries in the CONFIG.SYS file are used to control the downloading of applications via the ZONTALK 2000 download program:

*ZA=*xxxxx*	ZONTALK application ID
*ZP=*xxxxx*	ZONTALK download phone number. May use imbedded dialing control characters
*ZR=*n*	ZONTALK rate: 0=300, 2=1200, 3=2400 Default is 1200
*ZT=*xxxxx*	ZONTALK terminal ID

For LAN-type terminals, only *ZA and *ZT apply.

*ZX — Start Application Following Download

When *ZX=1, the application automatically starts following completion of a successful download (direct, ZONTALK, or terminal-to-terminal). If *ZX is set to any value besides 1, the terminal follows the normal sequence, whereby DOWNLOAD DONE is displayed and System Mode may be entered via a key press.

Application CONFIG.SYS Control

The library provides several functions for searching and updating CONFIG.SYS entries:

`get_env()` Retrieves a given environment variable and its value from the CONFIG.SYS file.

`put_env()` Stores an environment variable and its value in CONFIG.SYS. The following restrictions apply:

- ZONTALK 2000 only: Keys must be 7 bytes or less.
- Entries prefixed with an asterisk (*) are reserved for system use only.
- Do not use control codes (values between 0x00 and 0x1F).

`get_lan_config()` Retrieves the *LAD, *LBR, *LHA, and *LTW parameters stored in CONFIG.SYS, as well as four others not associated with CONFIG.SYS.

`set_env_buffer()` Sets the buffer where all environment variables returned by `getenv()` will be stored.

`getenv()` Searches CONFIG.SYS for an environment variable string, returning a pointer to the string associated with the environment variable.

If this function is used prior to the `set_env_buffer()` call, the system will allocate space for the data and return a pointer to this area. Since repeated use of this procedure can waste memory, use `get_env()` and

put_env() to conserve memory and aid readability.

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

Example

```
/* CONFIG.C */

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

char buffer[20];
main () {

   /* Add *ZA to CONFIG.SYS */
   put_env("*ZA", "APPLICATION", 11);

   /* Get the value of *ZA
      Use return length to make NULL-terminated string. */
   buffer[get_env("*ZA", buffer, 20)]=0;

   /* Display value of *ZA */
   printf("\f*ZA=%s",buffer);
   SVC_WAIT(2000);

   /* Add *ZP to CONFIG.SYS */
   put_env("*ZP", "555-1212",8);

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

   /* Put the value of "*ZP" into 'buffer' */
   getenv("*ZP");

   /* Display value of *ZP */
   printf("\f*ZP=%s",buffer);
}
```

C H A P T E R

5

Memory Management

The effective use of terminal RAM is an important issue for every application programmer. The more RAM needed to support an application and its data files, the more expensive the transaction system. Also, larger applications take longer to download, increasing long distance phone charges when enhancements or fixes must be sent out.

Application memory management should strive for two goals:

- Maximize file space available for application and data files.
- Ensure adequate space is available for stack and heap usage.

Using Terminal Memory

In the TXO environment, all data is stored in RAM, of which two types exist: volatile (non-banked) memory and non-volatile (banked). Non-volatile RAM is used to store the application program(s) and all data files. Volatile RAM is used for all global and local data, the

stack and heap, and is lost when the terminal is powered on or upon exit from System Mode.

Volatile Memory Management

A total of 64k of *volatile address space* is available for OMNI 300 Series terminals. While the operating system reserves 44k of volatile space for its own code, and 6k for operating system data, the remaining 14k is available for downloadable device drivers and volatile application data (global data).

Downloadable device drivers must be stored in the file DATA.EM; application data must be stored in the file SYSTEM.BIN.

DATA.EM This is the application data file, which contains the applications global data, local variables, the heap and the system stack. Application data is volatile variable storage and cannot be used for data files.

SYSTEM.BIN The SYSTEM.BIN file contains the application's downloadable device drivers. This file is always present on the system; if no drivers are present, its data size is zero bytes.

- *DATA.EM and SYSTEM.BIN together cannot exceed a total of 14k of address space.*
- *Application data will always use some space; estimate a minimum of 2k for any application.*
- *Any volatile (non-banked) address space not used for downloadable device drivers or application data is available for application code and data files.*

Non-volatile Memory Management

Application code file(s), data files and the communications buffer pool are stored in OMNI 300 Series *banked address space* (non-volatile memory). Also, any volatile memory not used by the files DATA.EM and SYSTEM.BIN, and the operating system itself, is available for application code, see *Volatile Memory Management*.

Non-volatile address space is shared memory—the less space used for the communications buffer pool, the more space available for the application and its data files.

The following table shows the minimum and maximum amount of address space available for non-volatile memory. Use this table as a guideline to application design.

	Terminal RAM					
	64k		128k		256k	
	Min.	Max.	Min.	Max.	Min.	Max.
Communications Buffer Pool	1k	8k	1k	8k	1k	8k
Code and Data Files *(Code file cannot exceed 64k)*	36k	55k	100k	119k	228k	247k
Contact your VeriFone representative regarding other RAM sizes.						

Tips on Managing Application Data

One of the most important design issues for the OMNI 300 Series terminal is the efficient use of the 2k – 14k application data memory region. The following tips should help you effectively manage this data space:

☞ *Check the values returned from TXO Workbench after the link/convert stage. For example, assume the following values are returned:*

```
code = 34681
data = 2386
stack = 3000 }..... (2386 + 3000 = 5386)
file size = 40067
```

Since the total amount of application data and stack space is less than 14k, and the "code" value is less than 64k, the application is well within the recommended limits.

☞ *Minimize the use of global variables. Unlike local variables, global variables are not dynamic. They always occupy space in the stack; local variables only occupy space in the stack when they are used, and also have smaller opcodes than global variables, therefore taking up less space.*

☞ *Avoid deep nesting of functions. Nesting function calls uses large amounts of stack space.*

☞ *Plan for the future. If you think you will use a downloadable device driver in the future, remember to save room for it.*

☞ *If the entire 14k is not used for application data or downloadable device drivers, the free RAM will be used by the file system.*

☞ *Allow a minimum of 2k for the application data, which will always use some space.*

☞ *Minimize the use of literals by moving them to data files where they can be better managed. Literals stored in the 14k data area decrease the amount of RAM available for stack and heap. Reading literal data from files into local variables is more efficient.*

☞ *Avoid using literals in #defines as they tend to use large amounts of stack space. When declared as a #define, the string is duplicated in memory each time it is referenced in a function call.*

☞ *Design your application to rely on downloaded data files and table-driven techniques to make it more flexible and easier to maintain and update.*

Managing Application Code

Calculating Available File Space

The amount of memory available for application code and data files varies, based on the following:

- Number of communications buffers
- RAM configuration of the terminal
- Usage of application data area

The amount of RAM available for code files and data files is calculated as follows:

Static RAM Size of Terminal	128k
Less OS Data Area	6k
Less Application Data Area	2k – 14k
Less Buffer Pool	1k – 8k
Remainder is available RAM for code and data files	100k – 119k

For example, in a 128k terminal where *B (number of buffers) has been set to 8, the stack and global data consist of 9k, and a 3k device driver is being used, the amount of available RAM for code and data files is 108k.

Static RAM	128k
Less OS Data Area	6k
Less Application Data Area	9k
Less Device Driver	3k
Less Comm. Buffers (8*256)	2k
Remainder is available RAM for code and data files	108k

These calculations are provided to help you estimate RAM usage. For a more accurate measurement, you can call the `dir_get_sizes()` function.

Communications Buffers

Communications buffers are used by the data communications device drivers (`MODEM,RS232,PIN PAD`). Each buffer is 256 bytes in length and can contain up to 254 bytes of data; two bytes are reserved—one for the count and one is unused. The maximum number of allocated buffers is 32, and the minimum number is 4.

The environment variable *B in CONFIG.SYS contains the number of buffers which will be allocated by the system. If *B does not exist, the minimum value of 4 will be used. Thus, the maximum amount of RAM which can be allocated for the communications buffer pool is 8k, with the minimum being 1k.

Lack of buffers may cause `write()` operations to fail. The return code from the `write()` operation to a communications device should always be checked. No matter how large you set your communications buffer pool size, the maximum effective block size the operating system may achieve by combining buffers is 4k. Thus `write()` operations of 4k can be achieved if that amount of free buffer space exists in the system.

Code and Data File Space

The application program is made up of one or more code files (*.OUT files). No code file can be more than 64K.

Data files are all other files used by the application including CONFIG.SYS, which contains the environment

variables. Files for batches, negative files, reports, etc., are stored in this area.

Managing Code Size

Managing the size of code files is an important application design issue. Below are important tips to help manage file size efficiently:

☞ *Monitor the* `CODE` *size returned from the link/convert stage of TXO Workbench. For initial implementation, the maximum suggested size of a code file is 45k, thus allowing room for future expansion of the code. Early design work can help determine if the application should be coded in multiple files (see Program Chaining and Application Control).*

☞ *Avoid using complicated or lengthy #defines. These cause a literal replacement of code in the file, increasing code size dramatically.*

☞ *Move literal data, such as prompts, to files where they can be compressed.*

☞ *When possible, use VeriFone's Application Construction Toolkit (ACT). This toolkit includes a set of library routines written and tested by VeriFone to be very efficient in both size and speed. Functions in the library leverage off of each other to help minimize code size.*

☞ *Avoid using complicated formatting and I/O functions such as:* `printf()`, `sprintf()`, `fscanf()`, *etc. These functions are large and slow. Instead, use low-level read and writes, or ACT functions. (Note: The examples found through this manual may include these functions for brevity and clarity. However, the finished application should use their low-level equivalents.)*

☞ *Use the Optimizers found in TXO Workbench.*

☞ *When the application is fully tested and debugged, use the TXO Workbench Compiler's -L option to remove debugging file names and line numbers, which increase code size by nearly 20%.*

Literal Strings

Avoid using literal strings whenever possbile. Each literal string results in a table entry, and this may create duplicate strings throughout the application. The preferred method of using strings for display data is to create a file containing all display messages.

The advantages of storing strings in a file and referencing them when needed are:

- Moves the storage requirement to the file space and out of the 14k limited space
- File compression can be used to further reduce the storage requirement
- Messages can be changed via a download

Avoid defining literal strings with `#define` statements, as they will use more memory. For example, a define of `"ENTER AMOUNT"` is the same as typing the literal string each time that the defined string is used. If a string will be used several times, it is more efficient to declare a variable and load it with the appropriate data.

Using the same example, a buffer containing the string `"ENTER AMOUNT"` could be declared and initialized in the application. Then the address of the buffer is referred each time the string is needed. While the string requires 12 bytes each time it is used, the address of the buffer holding the string is only two bytes.

☞ *When using global variables, do not initialize the value in the declaration. This eliminates start time overhead required by the operating system. Literals can be added to the message file, reducing the global variable requirement.*

Managing Data Structures

By properly managing data structures, you can reduce delays and make it possible to resume a transaction on a conditional basis after a power failure.

A C program may have hundreds or thousands of data elements—elements that are necessary to complete various application functions. In the TXO environment, all data is stored in RAM; as discussed earlier, two types of RAM exist: volatile and non-volatile. Non-volatile RAM is used to store the application program and all its data files. Volatile RAM is used for all global and local data, and the system stack and heap. All data in the memory area is lost when the terminal is powered on or upon exit from System Mode.

Initializing Global Data

Since global data is volatile, it must be initialized when the program starts. This initialization causes a delay that can be minimized by reducing automatic initialization (assignment of a value to a variable at the time it is declared) of global data. When the operating system starts, it must move initialization data to the indicated variables among other numerous tasks that must be performed. When creating executable code modules, the time required to transfer from one code module to the next is affected by the number of data elements to be initialized.

Large applications may require two or more executable code modules (via program chaining). To ensure there is minimal transfer delay, the application can initialize global data as part of the application by reading the data from a file directly to the global data elements. This is beneficial since the required data is stored in non-volatile memory and safe from power failures. The transfer time from files to data variables is very fast. The application also determines what initialization is required and the most convenient or appropriate time to complete the initialization. By maintaining this initialization file, it is possible for an application to resume a transaction on a conditional basis after a power failure.

Restoring Variable Data

C structures and arrays are a convenient way to save and restore large amounts of variable data using files. A structure may be declared that contains all the variables used for transactions. This may be saved to a file by treating the address of the structure as the buffer, and the size of the structure as the number of bytes to write. The data can be retrieved by reading from the file to the address of the structure. The number of bytes to read is the size of the structure.

General Considerations

Programmers should avoid using programming solutions purely according to the solution to similar problems in the past. Solutions should be examined to determine if they are the most appropriate ones for the problem, the language, the platform and the customer.

The TXO platform provides different solutions to program RAM space, file space, etc. The techniques required to take advantage of these platforms may be different than the techniques used on a PC or mainframe computer.

Every algorithm and process should be examined to ensure that the functionality provided is needed for the application. It can be more memory efficient to write a specialized function that provides only the processing needed by the application. Care should be used in making this decision when the function being replaced is a library routine. Many of the ACT Toolkit library routines use other functions in the library. This self-referencing further leverages the library code and reduces overall code space. When replacing a library function with a specialized function, both functions may be included in the application. This is because an engine or higher-level library function may need the function being replaced. For example, the `card_parse()` function calls `track_parse()`. Replacing `track_parse()` will result in additional code if `card_parse()` is used in the application.

Integration and Testing

Care should be taken to ensure memory constraints and issues are not allowed to present themselves late in the development process when solutions are more difficult to identify and quickly implement. Integration of the various modules should be performed often, and testing should be a continuous process. The application size and performance should be closely monitored throughout the development. The application should be designed with a conscious effort to develop executable code modules. This allows the use of the `SVC_RESTART()` function to branch to independent code modules and return upon completion.

C H A P T E R

6

File Management

The OMNI 300 Series terminal's file system provides a flexible and efficient way to store and retrieve data. The TXO libraries provide functions to access a number of file types, each suitable for certain application requirements.

When designing an application, select file types based on the data to be stored, and on the access requirements for that data. The following section describes each of the file access methods available and the type of data most suited to its use.

☞ *Design file management routines to minimize the need to increase or decrease file size. Reusing record space when possible instead of deleting and inserting new records can significantly improve the application's performance.*

File Conventions

File Storage

Files are stored in non-volatile (battery-backed) RAM. All files remain in memory even when power is removed from the terminal.

File Naming

Filenames may be up to 32 characters long and must be terminated by a NULL character. Any non-null character may be used in the name. Filenames are not case sensitive. For example, "TEST", "test", and "Test" are treated as the same file.

Default System Files

Four system files are normally present in the file system:

SYSTEM.BIN	Contains code for downloaded device drivers.
DATA.EM	Contains application global data, local variables, heap and system stack (also known as the application data file)
.OUT	The file extension used for all interpreted executable code files (example: MY_APPL.OUT).
CONFIG.SYS	Contains application environment variables, and keyed strings used by the operating system.

File Handles

In addition to opening any or all devices, up to 30 user files may be open at the same time. The system files listed previously do not count as part of the 30 open files.

TXO firmware lets you open the same file up to 30 times —useful when multiple file position indicators to the same file are needed. Each time the same file is opened, a new file handle is generated, and the remaining number of files that can be opened is decreased by one.

File Types and Access Methods

Generic Files

Generic files may contain any type of data, making them especially suitable for binary data. Data is accessed by byte address within the file so that any quantity of data may be read or written at any time, from any location within the file. In most applications, a fixed record length is used, typically based on the size of a data structure or union. With careful planning, variable length records may also be used.

Typically, a generic file will have a static record size determined by the total amount of space used by a data structure such as the one described below.

```
typedef  struct  MyRecord  MYRECORD;
struct  MyRecord
  {
  int record_type
  long first_field
  .
  .
  .
  int last_field
  };
#define   MyRecLen sizeof(MYRECORD)
```

Each record in the file is defined by the structure MyRecord; the record length is simply the size of the structure, defined by the constant MyRecLen. Using generic files with variable length records is discussed later.

Generic File Example

```
/* BINFILE.C */

/*  File demo for unstructured data. Shows correct syntax for
    using file functions. Program executes all statements and
    then terminates. */

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


main ()
   {
     int handle, bytes_written, bytes_read;
     int bytes_inserted, bytes_deleted, result;
     long int total_size, position;
     static char file_name[] = "TESTFILE.TXT";
     char buffer [13];
     strcpy (buffer,"123ABCefg");

/* Create (if it does not exist) the file TESTFILE.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 "123ABCefg" to TESTFILE.TXT - file pointer will
   be at end of file */
     bytes_written = write(handle,buffer,strlen(buffer));
     if (-1 == bytes_written) printf ("\fERROR # %d",errno);

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

```
/* Fill buffer with nulls then read 9 bytes from the current file
   pointer - "123ABCefg" will be put in buffer */
     memset (buffer,0,sizeof(buffer));
     bytes_read = read(handle,buffer,bytes_written);
     if  (-1 == bytes_read) printf ("\fERROR # %d",errno);

/* Place file pointer between "3" and "A" in file and insert "$$$"
   so file becomes "123$$$ABCefg" */
     result = lseek (handle,3L,SEEK_SET);
     if (-1 == result) printf ("\fERROR # %d",errno);
     bytes_inserted = insert (handle,"$$$",3);
     if (-1 == bytes_inserted) printf ("\fERROR # %d",errno);

/* Delete "ABC" from file to become "123$$$efg" - file pointer is
   currently between "$" and "A" due to above insert() call */
     bytes_deleted = delete (handle,3);
     if (-1 == bytes_deleted) printf ("\fERROR # %d",errno);

/*  Return the size of data portion of TESTFILE.TXT into position
    data portion equals 9 bytes */
     position = lseek (handle,0L,SEEK_END);

/*  Get the number of bytes in TESTFILE.TXT including overhead into
    the long int total_size. 4 will be returned to result */
     result = ioctl(handle, GetCtrl | 0, (char *)&total_size);

/*  Get the 12-byte file date/time into buffer.
    12 will be returned to result */
     memset (buffer,0,sizeof(buffer));
     result = ioctl (handle,GetCtrl | 1,buffer);

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

Variable Length Record (VLR) Files

VLR files allow data to be stored as records. The first byte in each record is a count byte (the number of bytes in the record, including the count byte), followed by data. Each record may have a maximum length of 254 bytes.

VLR files are particularly suited to ASCII data, and may contain arbitrary data, including embedded null characters. Data is accessed by record number rather than byte address, and requires that the file be processed from the beginning (first record then subsequent records searched until the correct record is found).

VLR files are best suited for chronological sequenced records with little or no random access requirements.

Compressed Variable Length Record (CVLR) Files

CVLR files are identical to VLR files with the addition of a compression algorithm applied to the data on writing and reading. This compression converts lower case characters to upper case, and stores numeric values (ASCII 0 – 9) as four bits. The compression and expansion of each record is handled by the file system, and is transparent to the application.

❖ *Byte values greater than 0x5F cannot be correctly translated when this compression is used, and should not be included in the file.*

Access time for CVLR files is somewhat slower than for VLR files. But, due to efficient compression programs, CVLR access time is still good. File space savings are especially noticeable on data files containing lots of numeric data.