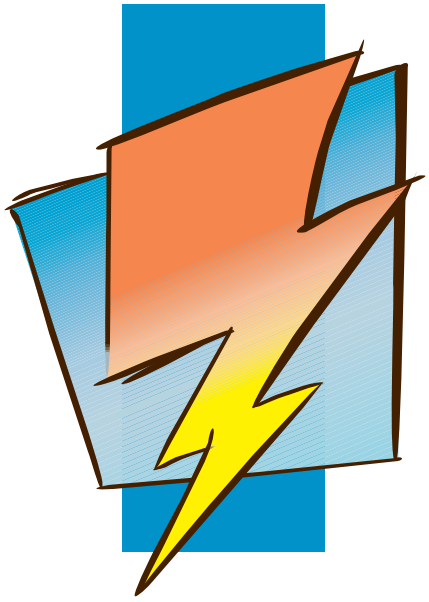
Open Watcom

CauseWay User's Guide



Version 2.0

Open Watcom

Notice of Copyright

Copyright © 2002-2021 the Open Watcom Contributors. Portions Copyright © 1984-2002 Sybase, Inc. and its subsidiaries. All rights reserved.

Any part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of anyone.

For more information please visit http://www.openwatcom.org/

Table of Contents

Introduction

1 Overview

CauseWay is a 386 DOS extender package for use with Open Watcom C/C++ programs. It is provided as a stub executable for which can be easily linked into DOS extended applications.

1.1 Introduction

Within the standard DOS, Windows and OS/2 DOS box environments, CauseWay supports 32-bit memory models for applications on PC compatibles with an 80386SX processor or above without the need to use overlays or crude stopgap measures such as EMS/XMS swapping. To do this, the DOS extender runs applications in protected mode, rather than the real mode normally used in the DOS environments. CauseWay supports both 16-bit and 32-bit protected mode applications operating under a DOS environment. It makes full use of 386-level chip capabilities including demand paging of code and data, variable-sized segments up to 4GB in length, mixing 16- and 32-bit segments as well as support for flat (non-segmented) memory addressing models. The CauseWay implementation of these powerful capabilities provides all their benefits while being transparent to the application user.

Applications created using CauseWay are compatible with the VCPI and DPMI standards and run equally well on systems with no protected mode drivers or programs. CauseWay applications work with such diverse environments as normal DOS, DesqView, Windows 3.0 and above in both standard and enhanced modes, as well as DOS windows within OS/2 2.0 and above, Windows 95 and above and Windows NT and later. CauseWay allocates memory from DPMI, VCPI, XMS, and INT 15H services, in addition to conventional DOS memory. This allows CauseWay applications to allocate memory through the CauseWay DOS extender without the need to detect or manipulate the various memory handling schemes.

A primary objective of CauseWay development was to ensure minimal effort would be needed by programmers to adapt their code to work with the CauseWay DOS e tender. As a result, most Open Watcom C/C++ and many realmode assembly language programs need no or minor changes to produce a fully operational CauseWay protected mode application.

1.2 Minimum System Requirements

CauseWay for Open Watcom C/C++ requires a 386SX based computer or better. The required operating environment is MS-DOS or PC-DOS 3.3 or higher, Windows 3.0 or higher, OS/2 2.0 and above, Windows 95 and above, Windows NT or higher or a compatible operating system that provides a DPMI or VCPI DOS environment.

CauseWay is to a large extent compatible with Tenberry Software's DOS/4GW. Most applications built for DOS/4GW will run with CauseWay without any changes.

1.3 CauseWay Memory Requirements

The recommended minimum amount of total free physical memory for CauseWay applications is 500KB total. 100-150KB of this memory must be conventional DOS memory, the remainder may be extended memory. CauseWay applications can run in less memory, down to the 300KB range, provided sufficient virtual (disk-based) memory is available, but application performance will decline significantly. More physical memory improves a program's performance, reducing virtual memory disk access overhead.

1.4 Operating CauseWay

When using CauseWay, simply follow the standard edit-compile/assemble-link programming cycle familiar to C and assembly language programmers.

Users compiling with WCL386 need to add the switch -l=CauseWay to the command line.

Example:

```
WCL386 -l=CauseWay myprog.c
```

This switch can be automated by adding -l#CauseWay to the **WCL386** environment variable, making CauseWay the default when compiling via WCL386.

Open Watcom users linking with WLINK should add the statement `system CauseWay` to the link command.

Example:

```
WLINK system CauseWay file myprog.obj
```

When running the DOS-extended application, DOS first loads the CauseWay DOS extender in conventional memory. CauseWay establishes the protected mode environment, retrieves the application from the executable file — loading it first into extended memory, then conventional memory if extended is exhausted, then virtual (disk-based) memory if conventional is exhausted — sets up the application for execution, and finally passes control to the application to begin operation. No additional files are required to make your application run in 386 protected mode using the CauseWay DOS extender.

`CWSTUB.EXE` will execute stand-alone LE-format files in the same way as `DOS4GW.EXE` does if the full file name, including extension, is listed after CWSTUB, e.g. `CWSTUB RUNME.EXE.` CWSTUB will override the extender bound to the application EXE, if any, with the CauseWay DOS extender version in `CWSTUB.EXE.`

1.5 Debugging Using WD

To debug CauseWay programs with the Open Watcom debugger after installing the CauseWay files, simply use the -tr=cw command line option.

Example:

```
WD -tr=cw myprog
```

This process can be automated by adding `-tr#cw` to your **WD** environment variable. Use the `set WD=tr#cw` command.

By default, CauseWay uses a Ctrl-Alt keypress to interrupt the WD debugger, rather than the Print-Screen key. This can be changed to any two, three, or four keypress value by modifying the ASCII file `CWHELP.CFG` at the BreakKeys line. See comments in this file for further detail. Note that a single keypress value will not work properly.

1.6 Operational Considerations When Using CauseWay

The **TEMP**, **TMP** and **CAUSEWAY=SWAP** environment variables are used by CauseWay to determine where to build its virtual memory swap file when an application is not operating under Windows or OS/2 (Windows and OS/2 provide their own virtual memory management). Since CauseWay has integrated virtual memory, disk space is considered part of total memory. If you use the **TEMP**, **TMP** or **SWAP** environment variable to point to a small RAM sk or almost full disk, free memory will be affected accordingly. If virtual (disk-based) memory is less than physical (installed on machine) memory, Cause Way turns off virtual memory. On the other hand, if you have a disk 300MB free, CauseWay will have no problem reporting 300MB free memory to your program, provided that virtual memory is not inhibited or limited by the **CAUSEWAY** environment variable memory settings.

Memory operates differently under Windows and OS/2. With OS/2, the DPMI setting for the session determines available memory. With Windows, available memory is the total of physical memory plus the swap file size less any memory already in use by Windows or another Windows application.

When creating a VMM swap file at application startup under DOS, CauseWay builds a list of possible paths in order of priority. CauseWay then works through the list until one of the entries provides both a valid drive and path specification and sufficient free space to being operation. The first entry to succeed becomes the swap file drive with no further processing of the list. If CauseWay reaches the end of the list without finding a valid drive, it disables the virtual memory manager. The order of priority is **CAUSEWAY=SWAP**, **TEMP**, **TMP** and application execution path.

If end users reboot the system or turn off power while executing a CauseWay application under DOS, a temporary file will be left on the system by CauseWay. This will usually be a zero length file unless the application was large enough to exceed physical memory and CauseWay had started using its virtual memory manager. The temporary file name is requested using standard DOS functions, meaning the name will vary with different versions of DOS. It typically is a mixture of letters and numbers with no extension, although .$$$ extension may be presented when operating under a network. Make sure you do not delete this temporary file while the CauseWay application is still active, as improper or erratic program operation, including lock-ups, may occur.

Application startup times may increase significantly if the free physical memory is less than the executable size. In such cases, not only must the executable be loaded into physical memory, but a virtual memory file of the executable file size must also be built. This file holds the portions of the executable that do not fit into physical memory and which have not been recently requested. After startup is complete, the program will operate normally, paging to and from virtual memory as necessary.

CauseWay automatically sets aside 32KB of low DOS memory for allocation and use by developer routines via the GetMemDOS API function. The 32K memory block is available even if CauseWay needs to use virtual memory just to load an application. The set-aside amount can be increased by using the

CAUSEWAY environment variable LOWMEM option, although the additional set-aside goal is not guaranteed to be reached if too little conventional memory is left for CauseWay's operating requirements.

1.7 Environment Variables

CauseWay can make use of three environment variables at runtime: **TEMP**, **TMP** and **CAUSEWAY**

1.7.1 TEMP and TMP Environment Variables

The **TEMP** and **TMP** environment variables specify the directory and drive where a swap file is built by CauseWay's virtual memory manager (VMM) when operating under DOS. Windows and OS/2 provide their own memory management functions which override CauseWay's use of the **TEMP** and **TMP** environment variables. The path indicated by **TEMP** will be used under DOS if both **TEMP** and **TMP** environment variables exist. Both settings are superseded by the **CAUSEWAY=SWAP** environment variable setting.

```
SET TMP=C:\SWAP
```

The example above directs the CauseWay DOS extender to create its swap file, if any, in the C:\SWAP directory.

If no **TEMP**, **TMP** and **CAUSEWAY=SWAP** settings are present or are invalid, the current drive is used when creating a swap file. If free drive space is less than physical memory (extended and conventional) available at startup, then the DOS extender VMM is disabled, no swap file is created, and virtual memory is not available to the application.

1.7.2 CAUSEWAY Environment Variable

The **CAUSEWAY** environment variable controls operation of the DOS extender at application runtime. Eleven (11) options are supported, although they are ignored in a Windows or OS/2 DPMI environment. Use any combination of the options in the following format:

```
SET CAUSEWAY=[<setting_1>;][<setting_2>;][<setting_n>;]
```

Items in square brackets ([ ]) are optional. Do not actually type the brackets if you use the optional items. Items in brackets (< >) should be replaced with actual values, separated by semicolons. Following is a description of the valid settings:

BIG1

Force CauseWay to use an alternate method to determine available extended memory under RAW memory environments (no DPMI host, no HIMEM.SYS loaded), allowing CauseWay to see more than 64MB of memory on machines which do not support more than 64MB under original INT 15h method. This method uses INT 15h function 0e801h to determine available extended memory, falling back to the original function if 0e801h fails. Note that old machines may not support this function and there is a slight chance that some older machines may not work if this setting is used.

DPMI

Force DPMI rather than default VCPI usage whenever possible (recommended for 386Max and BlueMax users). The memory manager must support DPMI or else this setting is ignored.

EXTALL

Force CauseWay to use all extended memory and sub-allocate memory from the bottom up instead of the default top-down approach. This setting is most useful for processor intensive environments which have a small hardware cache that does not cover the entire physical address range. Use of this setting means that no extended memory will be available for other programs while the application is loaded (including shelling to DOS).

HIMEM:<nnn>

Set maximum physical (conventional plus extended) memory that can be consumed by CauseWay where <nnn> is the decimal number of kilobytes that can be consumed. If memory allocation requests exceed this figure, CauseWay will use virtual memory, even if additional physical memory is present. If the HIMEM memory value exceeds available physical memory, then memory allocations operate normally. For example, HIMEM:2048 on a 4MB machine would force virtual memory use after 2MB of memory allocations (including loading the executable file). The remaining 2MB of memory could be used by other applications while the CauseWay application is active.

LOWMEM:<nnn>

Set DOS (conventional) memory to restrict it from use by CauseWay. This memory is in addition to the default 32KB low DOS memory block reserved by CauseWay for use by any applications which need to allocate DOS memory. <nnn> is the decimal number of kilobytes to reserve. If there is not enough conventional memory to satisfy the <nnn> request value, then CauseWay will leave all conventional memory free that is not required by the extender to operate. Note that this option does not guarantee the amount of free DOS memory, just how much needs to be free before CauseWay will consume DOS memory after exhausting all extended memory. For example, LOWMEM:200 will attempt to reserve 200KB of DOS memory, even if CauseWay has exhausted all extended memory and is using conventional memory to fill memory allocation requests.

MAXMEM:<nn>

Set maximum linear address space provided by CauseWay where <nn> is the decimal number of megabytes of linear address space. This setting is similar to HIMEM except that it includes any virtual memory. For example, MAXMEM:32 on a 16MB memory system restricts VMM disk space usage to 32MB, even if more disk space is present. MAXMEM:8 on the same system would restrict the application to 8MB of memory (all physical). Note that the setting is in megabytes, rather than kilobytes used in the LOWMEM and HIMEM options.

NAME:<filename>

Set a name, without a pathspec, to use the virtual memory temporary swap file. To set a path for the swap file, use the **CAUSEWAY=SWAP**, **TEMP**, or **TMP** environment variable. The filename must be valid, 12 characters or less. Additional characters are truncated or invalidate the filename, depending upon how DOS handles it (e.g., multiple periods make an invalid file name whereas a five-character extension is truncated to three). If the filename specified is invalid, CauseWay shuts off virtual memory. It makes no further attempts for a temporary file name. If a pre-existing file name is specified, CauseWay overwrites the file.

In conjunction with the PRE setting, the NAME setting can be a very powerful tool. Not only can no clusters be lost due to reset/reboot, but the leftover temporary file can be forced to a known name and location. Erase the swap file prior to running the application or leave it as a "permanent" swap file for CauseWay.

Note: In a multi-user or muti-CauseWay application situation, do not use the NAME setting unless it generates a unique file for each user and application. Otherwise, applications will be stepping on others' temporary files.

NOEX

Force CauseWay to not patch the INT 21h, function 4bh (EXEC) vector to turn off CauseWay's INT 31h extensions when the EXEC function is called. CauseWay normally turns off support of its INT 31h extensions with an EXEC call to be well behaved and avoid conflicts with other extenders or programs which may add their own extensions to INT 31h. However, if your CauseWay extended application shells out to DOS and passes the shelled-to application a callback address pointing to a routine within the parent CauseWay application, the callback will not work properly if the protected mode code uses the CauseWay extensions. With the NOEX setting present, CauseWay still supports its INT 31h extensions for those users who need to operate with callbacks in this fashion. Be aware that when the NOEX setting is present, CauseWay is less "well-behaved" about other programs which might add their own INT 31h extensions.

NOVM

Disable all virtual memory use by CauseWay. If physical memory is exhausted, CauseWay will fail further memory allocation requests.

PRE:<nnn>

Pre-allocates a swap file size, under non-DPMI environments, at start-up, where `<nnn>` is file size in megabytes, not kilobytes (same as MAXMEM).

There are at least two uses for this feature. First, to pre-allocate a virtual memory file size for applications with a total memory allocation (including EXE image) that does not exceed the set size. For example:

```
SET CAUSEWAY=PRE:4
```

pre-allocates a virtual memory file of 4MB. If an enduser resets or powers off the computer while the application is running and virtual memory is in use, the enduser's machine will not have lost clusters. There is only a 4MB temporary file to find and erase. If virtual memory usage exceeds 4MB, then SCANDISK must be used to recover lost clusters above and beyond what was pre-allocated.

Secondly, PRE can be used to allow your application to stake a claim to disk space before it needs it.

PRE may be used in conjunction with MAXMEM to ensure that virtual memory does not exceed the pre-allocation setting.

SWAP:<path>

Set CauseWay's virtual memory manager swap file path. This path takes precedence for choosing the location of a swap file over the **TEMP** and **TMP** environment variables.

Programmer's Guide

2 Using the flat memory model

Flat model is a non-segmented memory model that allows accessing data and executing code anywhere within the 4GB linear address region via a 32-bit offset without the need to use segment registers to point to the memory. In many respects, the memory model is identical to the tiny memory model well-known to DOS C and assembly language programmers, but supporting a memory region of 4GB rather than 64KB.

When using flat model, all linear addresses directly map to the physical address specified when an application is running in pure DOS environment. For example, writing to memory location 0B8000h will address video memory. Reading from memory locations in the 0FFF00h range will access ROM code. Under DPMI there is usually no direct relationship between linear and physical addresses, the DPMI host will however emulate access to memory below 1MB as appropriate without requiring any special considerations on the part of the application writer.

Generally speaking, flat is an improved version of the older near memory model supported in previous versions of CauseWay. Unlike near, flat model supports multiple segments, mixing 16- and 32-bit segments, and direct linear memory addresses without translation. Flat model supports all near memory model code without translation, including automatic handling of near-specific API functions.

3 Using DLLs with Open Watcom C/C++

It is recommended that you understand and familiarize yourself with the basic operation of DLLs (Dynamic Link Libraries) under Windows or OS/2 before using them with CauseWay under DOS. No attempt is made here to explain the fundamentals of DLL architecture and operation. You should also study the provided DLL example code.

DLL code should be compiled with the -s option to disable stack checking and the -bd option to generate DLL-suitable code. Specify a system type of `CWDLLR` for register-based parameter passing or `CWDLLS` for stack-based parameter passing.

A DLL file is a standard EXE file with the following requirement: The program start address should be an initialization and termination function, rather than a main program entry point. The entry address will be called twice: Once after loading to allow the DLL to perform initialization and once just prior to the DLL being unloaded from memory to allow it to clean up for termination. Entry conditions are: register EAX=0 for initialization and register EAX=1 for termination. An initialization code return value of EAX=0 indicates no errors. A code of any other value indicates an error has occurred and loading should be terminated. If an error condition is returned, it is up to the DLL to display an error message, CauseWay will simply report a load error. The entry address is a FAR call, so the initialization code should use a RETF to return control to the calling program.

A minimal DLL startup system is provided in the `DLLSTRTR.OBJ` (register-based) and the `DLLSTRTS.OBJ` (stack-based) files.

CauseWay loads DLLs when the program being loaded has references to external modules in the DLL. CauseWay searches the execution path for any DLL or EXE file (in that order) which has the proper internal module name. The module name is not used when searching. For example, a file named `USEME.DLL` contains a module named Spelling_Checker. The name of the module (Spelling_Checker) is set by the NAME option in your link file. If no NAME is specified, then the module's name will default to its file name without an extension. In this example, the module name would become USEME if no NAME is specified.

Following is an overview of the standard link file commands used to create DLLs. You may also refer to the Open Watcom Linker documentation for a description of these commands.

EXPORT function_name

where ***description***

function_name allows you to make a symbol (function name) available to other modules. It must be declared as a public symbol so that the linker can export it.

IMPORT [local_name] module_name.function_name

IMPORT [local_name] module_name.ordinal

where ***description***

local_name is an optional parameter. It is the symbol which the importing program references the function by, i.e. the symbol declared as external. If no local name is specified, then *function_name* is used.

module_name is the name of the module that contains the function. It is not the file name. IMPORT module names are resolved by searching DLL and EXE files for the correct module name.

function_name is the symbol by which the function is known in the EXPORTing module, i.e. the symbol that is declared as public.

ordinal_number functions can also be imported by number. This is the entry number in decimal, starting at 1, in the EXPORTing module's export table to link to. *local_name* must be specified when using ordinals, otherwise there is no symbolic reference to internally resolve.

In the DLL calling code, *local_name* and *function_name* need to be declared as external.

NAME module_name

where ***description***

module_name is a symbol by which the module should be identified when resolving IMPORT records in a calling program.

IMPORTed module names can contain a partial path. For example, DLL\spelling_checker would instruct the loader to look in <execution path>\DLL for a module with a name of spelling_checker.

Notes:

You must take the responsibility to make sure that the IMPORTed function or module calling conventions match the calling code. For example, the loader will load a 16-bit module to resolve an IMPORT in a 32-bit program without complaint. In this case, if the IMPORTing program does a 16-bit far call, then everything will work correctly, but a 32-bit call will fail unless the 16-bit DLL module ends with a 32-bit RET instruction.

Importing by function name may slow program performance if the symbols are frequently referenced. In such cases, consider using the ordinal command to speed up the access times because module name references are automatically converted to ordinals in an internal list which will only be processed once at load time.

4 Performance Considerations

There are a few ways to increase the operational efficiency of your CauseWay applications.

4.1 Memory Size

Almost without exception, the best way to increase runtime performance of all CauseWay applications is to ensure that physical memory is large enough to meet all of the program's needs. Performance suffers considerably when CauseWay creates a temporary file for virtual memory, swapping 4KB blocks of the program's code and data to and from disk. Naturally this may not be possible in all cases, but it is a worthy goal. Generally the more physical memory, even when virtual memory is being used, the better an application's performance.

When using a disk cache program, be sure not to use too much extended memory. Although a disk cache program is beneficial, allocating it too much memory can deprive CauseWay of required extended memory and can degrade application performance. However completely disabling disk cache will usually noticeably decrease performance as well. The optimal cache size depends on the particular application and computer system (amount of physical memory, disk I/O speed etc.) and there are no generally applicable "best" settings.

If your program uses virtual memory, CauseWay's VMM creates a temporary swap file. If you have more than one disk drive, then you may wish to direct creation of the swap file to the faster disk drive on your system using the **CAUSEWAY=SWAP** , **TEMP** or **TMP** environment variables. Do not create a RAM disk if this will lower your physical memory because this is less efficient than allowing CauseWay to use physical memory itself.

Remember that virtual memory is part of total memory when using CauseWay. If your default drive, or the drive pointed to by the **TEMP** or **TMP** environment variables has little free space, this will be reflected in total memory available to the CauseWay application. If disk free space is less than physical memory, then CauseWay shuts off all use of virtual memory. Windows and OS/2 handle virtual memory internally and supply it through the swap file and DPMI settings for the application.

4.2 Open Watcom C/C++ kbhit() replacement

Two optimized replacement versions of the Open Watcom C++ runtime library kbhit() function are provided. The files are KBHITR.OBJ and KBHITS.OBJ for, respectively, register-based and stack-based calling conventions. Simply link in the kbhit() replacement file appropriate for your compile options. These replacement routines bypass the normal INT checking of the keyboard and directly inspect the keyboard buffer to see if a keypress is pending. These routines significantly reduce the overhead in tight processing loops, which perform many kbhit()'s per second, by avoiding the interrupt call associated with checking for keystrokes.

Be aware that linking in the kbhitr or kbhits module means that the INT 28h idle call will not be made on kbhit() as normally occurs with the standard runtime library kbhit(). This may impact background

processing in applications which depend on INT 28h idle calls, such as the DOS PRINT utility which performs printing in the background, as well as operation under multitatsking environments.

4.3 DOS API Buffer Size

If your CauseWay application reads and writes files using large amounts of data on one read or write pass, you may wish to consider increasing the size of the internal DOS memory transfer buffer used by CauseWay. Refer to the SetDOSTrans and GetDOSTrans functions in the CauseWay API chapter for more information.

Note that the internal 8KB buffer is optimized for file transfers. Average file transfers of greater than 8KB will not necessarily improve performance with an increase in the buffer size. Generally speaking, the average file transfer must be 32KB or larger to gain any efficiency with an increased buffer size. Also, if you are using virtual memory, increasing the buffer size may actually slow down performance due to the decreased available physical memory. Test your application with both the default buffer and the desired new buffer size before permanently increasing the buffer size beyond the default.

4.4 API Memory Allocation

Inveterate tweakers may try out the SetMCBMax and GetMCBSize functions in the CauseWay API. These functions allow fine-tuning of the threshold used by CauseWay to allocate memory via a memory pool using memory control blocks (MCBs) rather than via normal DPMI functions. Since DPMI allocates memory in multiples of 4KB, setting the MCB threshold too low may result in a good deal of wasted memory and subsequent performance degradation.

5 Rules For Protected Mode Operation

The following information covers additional restrictions for protected mode compatible code that are not present when writing real mode compatible code.

Using protected mode rather than real mode requires following a few new programming rules to prevent processor faults from being generated, terminating the CauseWay application. These processor faults occur when an application breaks a protected mode programming rule.

Use the following rules for programming in protected mode:

1. A selector value (referred to as a segment value in real mode) loaded into a segment register references an area of memory that may occur anywhere within the machine's physical address space. The operating system can dynamically move this area of memory. Therefore, when dealing with selector values:

   • Never use segment registers as general purpose registers that can be loaded with arbitrary values. Every time a segment register is loaded with a value the processor checks the validity of the selector value and generates a fault if it is invalid.

   • Never perform segment arithmetic on a selector value. Segment arithmetic is usually performed in real mode code to either normalize a pointer, access a new paragraph of memory without changing an offset, or to access a single area of memory that is greater than 64KB in size. Since a selector value is an index into a table which contains the actual memory addresses, addition or subtraction of different selector values is meaningless and gives no useful results. (There exist special cases where contiguous selector values can be added or subtracted from for useful effect, but detailing these cases exceed the scope of this manual.)

   • Do not access data at an offset greater than the size of the associated selector. Attempts to do so result in a fault. This is one of the greatest strengths of protected mode because most obscure bugs in real mode code occur when a bad pointer value accesses the wrong area of memory, or when a buffer overflows and memory beyond the buffer is overwritten.

   • Do not attempt to write to or read from the NULL (zero value) segment. Attempts to do so results in a fault. In addition to valid selector values, segment registers can be safely loaded with a value of 0 but this selector value cannot be used to access memory or execute code.

2. Do not execute code in a data segment and do not write to data in the code segment. Use the CauseWay AliasSel function to map a data selector to the same physical memory area shared by a code selector when necessary. Even in this case, however, never write to memory using a CS: code segment override because it always causes a processor fault.

3. CauseWay handles most of the standard DOS interrupts transparently. When passing pointers to buffers for software interrupt calls not handled by CauseWay, create the buffers in low (conventional) DOS memory using the GetMemDOS function of the CauseWay API. In addition, convert the pointers from protected mode selector:offset pairs to real mode segment:offset pairs prior to the interrupt, and back upon your return from the interrupt.

6 CW.ERR File Information Format

The following information describes the format of the CW.ERR file that CauseWay creates if an exception occurs in a CauseWay application. This information can be very useful in tracking down exactly where and why an exception occurred.

6.1 Quick Reference Guide

If you do not know how to interpret assembly language or CPU instructions, and you want better detail on the location of an exception, you can frequently identify which routine caused the exception by cross-referencing the MAP file with the CW.ERR file. Following is a short guide to determine the offending routine.

Look at the value after the dash (-) listed for the CS segment register at the beginning of the ninth line in CW.ERR. This should be an eight digit number starting with several zeros. Now look at the MAP file of your application. Following the program, creation date and time lines in the MAP file is a listing of program segments showing their start, stop, length, name, class and count. Find the segment which has a start address equal to the eight digit number listed above. This is the entry for the program segment where the exception occurred.

Locate the public symbols listed by address in the MAP file. Each symbol in the program is listed in ascending address order. The address is composed of two values separated by a colon (:). Find the address group which begins with the eight digit number given above for CS segment register without the last digit. For example, if the CS eight digit number was 000205E0, look for an address beginning with 0000205E. If you cannot find any addresses beginning with the number, either no routines in the segment were declared public or else you have a version of the MAP file that was created at a different time than the application EXE file which generated the CW.ERR file.

If there is only one address beginning with the number, you have located the offending routine. If there is more than one address, examine the EIP value in CW.ERR. The EIP value is located in the middle of the seventh line immediately following the "EIP=" text. This value is the offset in the segment where the exception occurred. Find the symbol in the address group which has an address value following the ":" that is closest to the EIP but does not exceed the EIP value. This name of the symbol is the name of the routine which generated the exception. To continue our example, if you have the following four symbols of address 0000205E in the first half of the address line in the map file :

```
0000205E:000008DC       __DBFGOHOT

0000205E:00000944       __DBFGOCOLD

0000205E:00000986       __DBFGOTOID

0000205E:00000AEE       __DBFGOTOP
```

and the EIP value is 00000953, the closest routine name that does not exceed the EIP value in the second half of the address line is `__DBFGOCOLD`. Therefore, the exception occurred in the `__DBFGOCOLD` routine.

This method of locating the exception is not foolproof since it requires that the routine creating the exception in the program segment be declared public, but it should work for the majority of cases.

6.2 Sequence

The first line in CW.ERR is the CauseWay copyright message including the version of CauseWay used in the program. The version number may prove useful in tracking down problems that have been addressed in later versions of CauseWay.

The version is followed by the exception number and error code. These numbers, as well as all others in the CW.ERR file, are in hexadecimal. The values listed are those reported by the processor when the exception occurred. Detailed information about the significance of the values can be obtained in most 386 and above reference books. Typically the exception value will be 0DH, a General Protection Fault; 0CH, a stack fault usually due to stack overflow or underflow; or 0EH, a page fault due to improper memory reference. The error code is generally of little use for debugging purposes.

Next comes the general register values which indicate the state of the program when the exception occurred. The significance of register values is entirely dependent on the program being run at the time. CS:EIP register values can help track down the problem area by pinpointing exactly where in the code the exception occurred. Other register values may help determine why the exception occurred. In particular, look for use of registers as memory indices with values beyond the limit of the associated selector.

Next, the segment register values are displayed as a real selector value followed by the program relative value in bytes. If the second value is non-numeric (xxxxxxxx) then the segment register didn't contain a selector value allocated to the program at load time, although the value may be valid if it was dynamically allocated by the operating program. If there is a second value, it also appears in the program's .MAP file as the segment start address. This shows which segment a segment register is pointing to at the time of the exception. The CS (Code Segment) register points to the segment containing the code which is executing. The EIP register value indicates the offset within the CS segment where the exception occurred. With these two values, you can not only determine the segment, but the routine within the segment closest to where the exception occurred.

Segment register values are also useful in determining why an exception occurred. One common error is using an invalid selector value in DS, ES, FS, or GS. A segment register value of zero does not automatically indicate problem, but will cause a GPF if used to read or write to memory. In particular, be highly suspicious of DS and ES segment register values of 0000-xxxxxxxx since they are almost constantly used to to access memory. A zero value in DS or ES usually indicates a bad (NULL) memory pointer passed to a routine.

Next, the processor control register values are listed. These registers are unlikely to be of much use for debugging and will only be filled in when not running under a true DPMI host. For an exception 0Eh (page fault), CR2 is the linear address that was accessed for which no memory was mapped in. This may help track down the problem area.

Info Flags comes next. This value is returned by CauseWay's Info API function. Check it against the documentation for Info in the CauseWay API chapter to determine some aspects of the environment in which the program was running when the exception occurred, e.g. whether a DPMI host was being used.

Program Linear Load Address follows Info Flags. This value is th executable's load address in linear memory. It corresponds to one of the linear memory block entries described later.

In flat models, the EIP value minus the program linear load address is the address offset of the faulting location relative to the start of the program.

Following Program Linear Load Address is a display of the next 128 byte values at the CS:EIP location when the exception occurred. These are the hexadecimal byte values of the CPU instructions at the time of the exception. 386 reference books or some debuggers can be used to reconstruct the instruction operation codes that correspond to these hexadecimal byte values.

The SS:ESP displays follows CS:EIP. This display shows the last 128 bytes values stored on the CPU stack.

SS:EBP is next. It shows 128 byte values of the current stack frame negatively and positively offset from the current EBP value. C and other high level language routines use the EBP register to reference parameters passed on the stack and this display can show which parameters were passed.

The resource tracking details come next. Selectors are listed with the following headings:

```
sel  base  limit  type  D  mem  count
```

where	***description***
sel	Selector value.
base	Linear base address of selector.
limit	Limit of selector.
type	CODE or DATA.
D	16 or 32 to signify segment D bit.
mem	Y or N to indicate if the selector has a memory block associated with it.
count	segment count in MAP file, xxxx if dynamically allocated.

The selector list is finished off with a display of the total number of selectors allocated to the program. For example:

```
Total selectors: 0107
```

Linear memory blocks are listed following the selector list, and contain the following headings:

```
handle  base  length
```

where	***description***
handle	Linear memory [de]allocation uses handles to control the blocks. This field is the block's handle.
base	Linear base address of the memory block.
length	Length of the block in bytes.

The linear memory list ends with a display of the total linear memory allocated to the program, the real (rounded to 4KB pages) memory allocated in parentheses, and finally the total number of memory blocks. For example:

```
Total Linear memory: 000FEAC9 (001AA000) in 00000103 blocks
```

Entries in the selector list that have Y under "mem" should have a corresponding entry in the linear memory list.

Linear memory locks are listed after the linear memory block list and contain the following headings:

```
base  length
```

where ***description***

base Linear base of the locked region.

limit Length of the locked region.

Note: These values are passed by the program but the actual values have the base rounded down a page and the length rounded up a page to match 4KB boundary restrictions on locking. The values are reported in CW.ERR without using a rounded format to make it easier to cross reference this list to the other lists in CW.ERR.

Next, protected mode interrupt vectors are listed with the following headings:

```
No  sel  offset
```

where ***description***

No Vector number.

sel Selector value for handler.

offset Offset value for handler.

This information allows a cross-reference with the other lists to ensure CauseWay application installed handlers have been properly made.

Next, protected mode exception vectors are listed using the same format as protected mode interrupts.

After the protected mode exception vector list, real mode interrupt vectors are listed. They are shown in the same format as protected mode interrupts although the selector values are real mode segment values.

Only those interrupt and exception vectors altered by the program will be listed.

Lastly, Callbacks are listed in the CW.ERR file. They list all active Callbacks for the active application at the time of its termination.

CauseWay Reference

7 CauseWay Services

The following information describes the services available through CauseWay for low-level protected mode compatible functions and interrupt servicing.

The CauseWay services support both 16- and 32-bit selectors. Use of 32-bit selectors allows developers to directly access many megabytes of memory in a CauseWay program using only one selector value. In addition to the normal segment registers used in real mode programs, the additional segment registers FS and GS are always available for use by developers to access memory. Refer to a 386, 486, Pentium or compatible CPU reference book or manual for more information on 386+ level registers and instructions.

7.1 Internal Operation

A valid protected mode selector:offset is placed in the PSP at offset 34h for the file handle list pointer. Note that the default value in the program's PSP will point to the real mode PSP, not the protected mode PSP, even if the handle count is less than or equal to twenty. Code that makes use of the handle list should use the address at PSP+34h rather than assuming the list's position within the PSP. Also, when an application is operating in non-DPMI conditions the handle table will have an entry for CauseWay's VMM swap file.

The GetMem and GetMem32 calls with CX:DX and ECX set to -1 will report the largest free memory block, rather than total free memory. This value may be substantially lower than total free memory due to fragmentation of the linear memory blocks. Set CX to -1 (0FFFFH) and DX to -2 (0FFFEH) or ECX to -2 (0FFFFFFFEH) for GetMem and GetMem32, respectively, for the total free memory value.

7.2 Functions

The CauseWay functions are based on the DPMI specification and offer additional enhancements. This means that several of the DPMI 1.0 functions are available to the CauseWay programmer in all systems providing access to lower level functions should you need them. All DPMI 0.9 functions are always available.

7.3 CauseWay API

CauseWay provides an API for C and assembly language programmers as an extension of the DPMI API via INT 31h. Including the file CW.INC allows easy access to this API through appropriately named functions. You may also call the CauseWay API directly with appropriate register setup.

CauseWay also provides all of the DPMI 0.9 API services on systems without a true DPMI server (thus CauseWay itself is the DPMI host in such situations).

Some of the API services require pointers to a real mode register list. The format of this list follows:

```
dword     EDI
dword     ESI
dword     EBP
dword     Reserved
dword     EBX
dword     EDX
dword     ECX
dword     EAX
word      Flags
word      ES
word      DS
word      FS
word      GS
word      IP
word      CS
word      SP
word      SS
```

The values are passed to the target routine without any interpretation of their contents. There is no need to set the high words of the extended register entries unless the target routine requires them.

7.4 Stack Frames

Stack frames for 16-bit interrupts are the same as for real mode.

The stack frame for 16-bit exceptions follows:

```
word SS
word SP       - Original stack address
word Flags
word CS
word IP       - Original Flags:CS:IP values
word Err Code - Processor supplied exception error code
word CS
word IP       - Return address, returns to
                interrupt/exception dispatch code
```

7.5 Default API

The default CauseWay API follows. Functions that include the text `near` are intended only for backwards compatibility with CauseWay's near memory model. This model is now obsolete. The assembly language include file *CW.INC* also contains this list.

7.5.1 API functions (numerical index)

FF00 Info Get system selectors/flags

FF01 IntXX Simulate real mode interrupt

FF02 FarCallReal Simulate real mode far call

FF03 GetSel	Allocate a new selector
FF04 RelSel	Release a selector
FF05 CodeSel	Make a selector execute/read type
FF06 AliasSel	Create a read/write data selector from source selector
FF07 GetSelDet	Get selector linear base and limit
FF08 GetSelDet32	Get selector linear base and limit
FF09 SetSelDet	Set selector linear base and limit
FF0A SetSelDet32	Set selector linear base and limit
FF0B GetMem	Allocate a block of memory
FF0C GetMem32	Allocate a block of memory
FF0D ResMem	Resize a previously allocated block of memory
FF0E ResMem32	Resize a previously allocated block of memory
FF0F RelMem	Release memory allocated by either GetMem or GetMem32
FF10 GetMemLinear	Allocate a block of memory without a selector
FF11 GetMemLinear32	Allocate a block of memory without a selector
FF12 ResMemLinear	Resize a previously allocated block of memory without a selector
FF13 ResMemLinear32	Resize a previously allocated block of memory without a selector
FF14 RelMemLinear	Release previously allocated block of memory (linear address)
FF15 RelMemLinear32	Release previously allocated block of memory (linear address)
FF16 GetMemNear	Deprecated - Allocate an application relative block of memory
FF17 ResMemNear	Deprecated - Resize a previously allocated application relative block of memory
FF18 RelMemNear	Deprecated - Release previously allocated application relative block of memory
FF19 Linear2Near	Deprecated - Convert linear address to application relative address
FF1A Near2Linear	Deprecated - Convert application relative address to linear address
FF1B LockMem	Lock a region of memory
FF1C LockMem32	Lock a region of memory
FF1D UnLockMem	Unlock a region of memory

FF1E UnLockMem32	Unlock a region of memory
FF1F LockMemNear	Deprecated - Lock a region of memory using application relative address
FF20 UnLockMemNear	Deprecated - Unlock a region of memory using application relative address
FF21 GetMemDOS	Allocate a region of DOS (conventional) memory
FF22 ResMemDOS	Resize a block of DOS (conventional) memory allocated with GetMemDOS
FF23 RelMemDOS	Release a block of DOS (conventional) memory allocated by GetMemDOS
FF24 Exec	Run another CauseWay program directly
FF25 GetDOSTrans	Get current address and size of the buffer used for DOS memory transfers
FF26 SetDOSTrans	Set new address and size of the buffer used for DOS memory transfers
FF27 GetMCBSize	Get current memory control block (MCB) memory allocation block size
FF28 SetMCBSize	Set new MCB memory allocation block size
FF29 GetSels	Allocate multiple selectors
FF2A cwLoad	Load another CauseWay program as an overlay
FF2B cwcInfo	Validate and get expanded length of a CWC'ed file
FF2C GetMemSO	Allocate a block of memory with selector:offset
FF2D ResMemSO	Resize a block of memory allocated via GetMemSO
FF2E RelMemSO	Release a block of memory allocated via GetMemSO
FF2F UserDump	Setup user-defined error buffer dump in CW.ERR
FF30 SetDump	Disable/enable error display and CW.ERR creation
FF31 UserErrTerm	Call user error termination routine
FF32 CWErrName	Change error file name, with optional drive/pathspec
FFF9 ID	Get CauseWay identifier, PageDIRLinear and Page1stLinear info
FFFA GetPatch	Get patch table address
FFFB cwcLoad	Load/Expand a CWC'ed data file into memory
FFFC LinearCheck	Check linear address of memory
FFFD ExecDebug	Load CauseWay program for debug
FFFE CleanUp	Close all open file handles

7.5.2 API functions (alphabetical order)

AliasSel Create a read/write data selector from source selector.

Inputs: AX= 0ff06h
BX= Source selector

Outputs: Carry set on error, else
AX= New data selector

Errors: If an invalid selector is passed in BX, this function returns with carry set.

Notes: This function always creates a read/write data selector regardless of the source selector type. It can be used to provide write access to variables in a code segment.

CleanUp Close all open file handles.

Inputs: AX= 0fffeh

Outputs: None.

Errors: None.

CodeSel Make a selector execute/read type.

Inputs: AX= 0ff05h
BX= Selector
CL= Default operation size. (0=16-bit,1=32-bit)

Outputs: Carry set on error.

Errors: If an invalid selector is passed in BX, this function returns with carry set.

Notes: This functions allows a selector to be converted to a type suitable for execution.

cwcInfo Validate and get expanded length of a CWC'd file.

Inputs: AX= 0ff2bh
BX= File handle.

Outputs: Carry set if not a CWC'd file, else
ECX= Expanded data size.

Errors: None.

Notes: The file pointer is not altered by this function.

cwcLoad Load/Expand a CWC'ed data file into memory.

Inputs: AX= 0fffbh
BX= Source file handle. ES:EDI= Destination memory.

Outputs: Carry set on error and EAX is error code, else
ECX= Expanded data length.

Errors: 1 - Error during file access.
2 - Bad data.
3 - Not a CWC'ed file.

Notes: The source file's file pointer doesn't have to be at zero. A single file might be several CWC'ed files lumped together, as long as the file pointer is moved to the right place before calling this function.
If error codes 1 or 2 are reported then the file pointer will be wherever it was last moved to by this function. For error code 3 the file pointer will be back at its original position on entry to this function. If no error occurs then the file pointer will be moved to whatever comes after the compressed data.

CWErrName Change error file name, with optional drive/pathspec.

Inputs: AX = 0ff32h
CX:[E]DX = selector:offset of ASCIIZ error file name

Outputs: None

Errors: None

Notes: If the error file name is invalid when a fault occurs, CauseWay defaults to using the standard CW.ERR file name in the current directory. The file name including any path and drive must not exceed 80 characters or it will be truncated. CX:EDX are not checked for validity and passing invalid values may cause a fault within the DOS extender. The ASCIIZ name pointed to by CX:EDX is copied to an internal DOS extender location and may be safely modified after calling the CWErrName function.

cwLoad Load another CauseWay program as an overlay.

Inputs: AX= 0ff2ah
DS:EDX= File name.

Outputs: Carry set on error and AX = error code, else
CX:EDX= Entry CS:EIP
BX:EAX= Entry SS:ESP
SI= PSP.

Errors: 1 - DOS file access error.
2 - Not recognisable file format.
3 - Not enough memory.

Notes: Program is loaded into memory, but not executed.
The PSP returned in SI can be passed to RelMem to release the loaded programs memory and selectors. Only the memory and selectors allocated during loading will be released, it is the programs responsability to release any additional memory etc allocated while the program is running. Alternatively, if you pass the PSP value to INT 21h, AH=50h before makeing additional memory requests and then reset to the origional PSP the memory allocated will be released when the PSP is released.

Exec Run another CauseWay program directly.

Inputs: AX= 0ff24h
DS:[E]DX= File name.
ES:[E]SI= Command line. First byte is length, then real data.
CX= Environment selector, 0 to use existing copy.

Outputs: Carry set on error and AX = error code, else
AL=ErrorLevel.

Errors: 1 - DOS file access error.
2 - Not recognisable file format.
3 - Not enough memory.

Notes: Only the first byte of the command line (length) has any significance to CauseWay so you are not restricted to ASCII values. It is still stored in the PSP at 80h though so the length is still limited to 127 bytes.

ExecDebug Load CauseWay program for debug.

Inputs: AX= 0fffdh
DS:EDX= File name.
ES:ESI= Command line. First byte is length, then real data.
CX= Environment selector, 0 to use existing copy.

Outputs: Carry set on error and AX = error code, else
CX:EDX= Entry CS:EIP
BX:EAX= Entry SS:ESP
SI= PSP.
DI= Auto DS.
EBP= Segment definition memory.

Errors: 1 - DOS file access error.
2 - Not recognisable file format.
3 - Not enough memory.

FarCallReal Simulate real mode far call.

Inputs: AX= 0ff02h
ES:[E]DI= Real mode register list.

Outputs: Register list updated.

Errors: None.

Notes: This function works much the same as IntXX but provides a 16 bit FAR stack frame and the CS:IP values are used to pass control to the real mode code.

GetCallBack Allocate real mode callback address.

Inputs: AX= 0303h
DS:[E]SI= Call address.
ES:[E]DI= Real mode register structure.

Outputs: Carry set on error, else
CX:DX= Real mode address to trigger mode switch.

Errors: Callbacks are a limited resource. Normally only 16 are available per virtual machine. Use them carefully and release them as soon as they are no longer required.

Callback: Interrupts disabled.
DS:[E]SI = Selector:Offset of real mode SS:SP.
ES:[E]DI = Selector:Offset of real mode call structure.
SS:[E]SP = Locked protected mode stack.
All other registers undefined.
To return from callback procedure, execute an IRET to return.
ES:[E]DI = Selector:Offset of real mode call structure to restore.

Notes: Real mode callbacks provide a means of switching from real mode to protected mode. This function returns a unique real mode address that when given control in real mode, switches to protected mode and passes control to the protected mode routine supplied at entry to this function. On entry to the protected mode code the real mode register structure contains all the real mode register values.

GetDOSTrans Get current address and size of the buffer used for DOS memory transfers.

Inputs: AX = 0ff25h

Outputs: BX = Real mode segment of buffer.
DX = Protected mode selector for buffer.
ECX = Buffer size

Errors: None

Notes: This buffer is used by the INT API translation services, e.g., INT 21h, AH=40h (write to file). The default buffer is 8K and uses memory that would otherwise be wasted. This default is sufficient for most file I/O but if you are writing a program that reads/writes large amounts of data you should consider allocating your own larger buffer and pass the buffer's address to CauseWay to speed file I/O.

GetEVect Get Protected mode exception handler address.

Inputs: AX= 0202h
BL= Exception vector number.

Outputs: Carry set on error, else
CX:[E]DX= selector:offset of handler.

Errors: The number in BL must be in the range 0-1Fh. Anything outside this range returns carry set.

GetMCBSize Get current memory control block (MCB) memory allocation block size.

Inputs: AX = 0ff27h

Outputs: ECX = Current threshold

Errors: None

Notes: See SetMCBMax

GetMem Allocate a block of memory.

Inputs: AX= 0ff0bh
CX:DX= Size of block required in bytes. (use -1:-1 to get maximum memory size)

Outputs: Carry set on error, else
BX= Selector to access the block with or if CX:DX was -1,
CX:DX= size of largest block available.

Errors: The amount of memory available is limited by physical memory present and free disk space of the drive being used by the VMM. If CauseWay is unable to find a large enough block this function returns carry set.

Notes: This function allocates a block of extended (application) memory and allocates a selector with a suitable base and limit.

GetMem32 Allocate a block of memory.

Inputs: AX= 0ff0ch
ECX= Size of block required in bytes. (-1 to get maximum memory size)

Outputs: Carry set on error, else
BX= Selector to access the block with or if ECX was -1,
ECX= size of largest block available.

Errors: See GetMem

Notes: This function allocates a block of extended (application) memory and allocates a selector with a suitable base and limit.

GetMemDOS Allocate a region of DOS (conventional) memory.

Inputs: AX= 0ff21h
BX= Number of paragraphs (16 byte blocks) required.

Outputs: Carry set on error and BX= largest block size,
AX=DOS error, else
AX= Initial real mode segment of allocated block
DX= Initial selector for allocated block

Errors: If there are not enough selectors or memory available then this function returns carry set.

Notes: If the size of the block requested is greater than 64KB bytes (BX >1000h) then contiguous descriptors are allocated. If more than one descriptor is allowed under 32-bit applications, the limit of the first descriptor is set to the size of the entire block. All subsequent descriptors have a limit of 64KB except for the final descriptor which has a limit of block size modulo 64KB. For 16-bit applications, always set the limit of the first descriptor to 64KB.

GetMemLinear Allocate a block of memory without a selector.

Inputs: AX= 0ff10h
CX:DX= Size of block required in bytes.

Outputs: Carry set on error, else
SI:DI= Linear address of block allocated.

Errors: If not enough memory is available to satisfy the request then this function returns carry set.

Notes: Addresses returned by this function may be above 16MB.

GetMemLinear32 Allocate a block of memory without a selector.

Inputs: AX= 0ff11h
ECX= Size of block required in bytes.

Outputs: Carry set on error, else
ESI= Linear address of block allocated.

Errors: See GetMemLinear

Notes: Addresses returned by this function may be above 16MB.

GetMemSO Allocate a block of memory with selector:offset.

Inputs: AX = 0ff2ch
CX:DX = Size of block required in bytes

Outputs: Carry set on error, else
SI:DI = selector:offset of allocated memory

Errors: See GetMem

Notes: This function allocates a block of memory with an associated selector:offset. The allocation does not consume a selector on each call as does GetMem because a non-zero offset from an existing selector for this allocation type is returned. GetMemSO is useful for applications which make a large number of allocations where running out of selectors with GetMem could be a problem. A potential drawback is that memory accesses beyond the allocation boundary may go undetected since the selector is shared among several allocations. Also, resizing the block can change the selector:offset of the block.

GetPatch Get patch table address.

Inputs: AX= 0fffah

Outputs: EDX= Linear address of patch table.

Errors: None.

GetRVect Get real mode interrupt handler address.

Inputs: AX= 0200h
BL= Interrupt vector number.

Outputs: CX:DX= selector:offset of handler.

Errors: None.

GetSel Allocate a new selector.

Inputs: AX= 0ff03h

Outputs: Carry set on error, else
BX= Selector.

Errors: Approximately 8192 selectors are available initially. While this is a relatively large quantity, it is obviously possible to run out if the system is heavily loaded or selectors are being wasted.

Notes: A selector is allocated and initialized with a base of 0, a limit of 0 and as read/write expand up data. Use SetSelDet to make the selector useful, setting an appropriate base and limit.

GetSelDet Get selector linear base and limit.

Inputs: AX= 0ff07h
BX= Selector

Outputs: Carry set on error, else
CX:DX= Linear base.
SI:DI= Byte granular limit.

Errors: If an invalid selector is passed in BX, this function returns with carry set.

GetSelDet32 Get selector linear base and limit.

Inputs: AX= 0ff08h
BX= Selector

Outputs: Carry set on error, else
EDX= Linear base.
ECX= Byte granular limit.

Errors: If an invalid selector is passed in BX, this function returns with carry set.

GetSels Allocate multiple selectors.

Inputs: AX= 0ff29h
CX= Number of selectors.

Outputs: BX= Base selector.

Errors: None.

Notes: The selectors are allocated and initialised with a base of 0, a limit of 0 and as read/write expand up data. Use SetSelDet to make the selectors useful.

GetVect Get Protected mode interrupt handler address.

Inputs: AX= 0204h
BL= Interrupt vector number.

Outputs: CX:[E]DX= selector:offset of handler.

Errors: None.

ID Get CauseWay identifier, PageDIRLinear and Page1stLinear info.

Inputs: AX= 0fff9h

Outputs: ECX:EDX= CauseWay identifies.
ESI= Linear address (PageDIRLinear)
EDI= Linear address (Page1stLinear)

Errors: None.

Info Get system selectors/flags.

Inputs: AX= 0ff00h

Outputs: AX= Selector for real mode segment address of 00000h, 4GB limit.
BX= Selector for current PSP segment. 100h limit.
[E]CX= DOS transfer buffer size. Always <64KB.
DX= DOS transfer buffer real mode segment address.
ES:[E]SI= DOS transfer buffer address.
ESI+ECX always <64KB
EDI= System flags. Bits significant if set.
0 - 32 bit code default.
1 - Virtual memory manager functional.
2-3 - Mode, 0 - raw, 1 - VCPI, 2 - DPMI.
4 - DPMI available.
5 - VCPI available.
6 - No memory managers.
7 - Descriptor table type. 0 - GDT, 1 - LDT.

Errors: None

Notes: Bits 1-2 of DI indicate the interface type being used by CauseWay.

Bits 4-5 indicate the interface types that are available. Bit 7 indicates the descriptor table being used to allocate selectors to the application when on a raw/VCPI system. The DOS transfer buffer is the area CauseWay uses to transfer data between conventional and extended memory during DOS interrupts. This memory can be used as temporary work space to access real mode code as long as you remember it may be overwritten the next time you issue an INT in protected mode that requires segment pointers.

IntXX Simulate real mode interrupt.

Inputs: AX= 0ff01h
BL= Interrupt number.
ES:[E]DI= Real mode register list.

Outputs: Register list updated.

Errors: None.

Notes: The real mode register list referenced by ES:[E]DI should contain the register values you want passed to the real mode interrupt handler. CauseWay fills in the SS:SP and Flags values to ensure that legal values are used and the CS:IP entries are ignored. This function bypasses protected mode interrupt handlers and provides access to INT APIs that require segment pointers.

LinearCheck Check linear address of memory.

Inputs: AX= 0fffch
ESI= Linear address of memory.

Outputs: Carry set on invalid memory address.

Errors: None.

LockMem Lock a region of memory.

Inputs: AX= 0ff1bh
BX:CX= Starting linear address of memory to lock.
SI:DI= Size of region to lock in bytes.

Outputs: Carry set on error.

Errors: If any of the region specified is invalid or if not enough physical memory is available to fill the region specified, then none of the memory is locked and this function returns carry set.

Notes: Memory that is locked cannot be swapped to disk by the VMM. Locking applies to memory on page (4KB) boundaries. Therefore, areas of memory below and above the memory being locked are locked if the specified region is not aligned to a page boundary.

LockMem32 Lock a region of memory.

Inputs: AX= 0ff1ch
ESI= Starting linear address of memory to lock.
ECX= Size of region to lock in bytes.

Outputs: Carry set on error.

Errors: See LockMem.

Notes: Memory that is locked cannot be swapped to disk by the VMM. Locking applies to memory on page (4KB) boundaries. Therefore, areas of memory below and above the memory being locked are locked if the specified region is not aligned to a page boundary.

RelCallBack Release a real mode callback entry.

Inputs: AX= 0304h
CX:DX= Real mode address returned by GetCallBack

Outputs: None.

Errors: None.

Notes: Uspe this function to release callback addresses once they are no longer needed.

RelMem Release memory allocated by either GetMem or GetMem32.

Inputs: AX= 0ff0fh
BX= Selector for block to release.

Outputs: Carry set on error.

Errors: If an invalid selector is passed in BX or the memory was not allocated via GetMem or GetMem32, this function returns carry set.

RelMemDOS Release a block of DOS (conventional) memory allocated by GetMemDOS.

Inputs: AX= 0ff23h
DX= Selector of block to free.

Outputs: Carry set on error and AX= DOS error code.

Errors: If an invalid block is passed, this function returns carry set.

Notes: All descriptors allocated for the memory block are automatically freed and therefore should not be accessed once the block is freed by this function.

RelMemLinear Release previously allocated block of memory (linear address).

Inputs: AX= 0ff14h
SI:DI= Linear address of block to release.

Outputs: Carry set on error.

Errors: If the address passed in SI:DI is not a valid memory block, this function returns carry set.

RelMemLinear32 Release previously allocated block of memory (linear address).

Inputs: AX= 0ff15h
ESI= Linear address of block to release.

Outputs: Carry set on error.

Errors: See RelMemLinear

RelMemSO Release a block of memory allocated via GetMemSO.

Inputs: AX = 0ff2eh
SI:DI = Selector:offset of block to release

Outputs: Carry set on error.

Errors: If an invalid selector:offset is passed in SI:DI or the memory was not allocated via GetMemSO, then this function returns carry set.

RelSel Release a selector.

Inputs: AX= 0ff04h
BX= Selector.

Outputs: Carry set on error.

Errors: If an invalid selector is passed in BX, this function returns with carry set.

Notes: Use this function to release selectors allocated by GetSel or AliasSel.

ResMem Resize a previously allocated block of memory.

Inputs: AX= 0ff0dh
BX= Selector for block.
CX:DX= New size of block required in bytes.

Outputs: Carry set on error.

Errors: If an invalid selector is passed in BX or not enough memory is available when increasing the block size, then this function returns carry set.

Notes: If the memory block cannot be resized in its current location, but a free block of memory of the new size exists, the memory is copied to a new block and the old one is released. The application is not affected as long as only the selector originally allocated with GetMem accesses the memory.

ResMemSO Resize a block of memory allocated via GetMemSO.

Inputs: AX = 0ff2dh
SI:DI = Selector:offset of block to resize
CX:DX = New size of block required in bytes

Outputs: Carry set on error, else
SI:DI = selector:offset of new memory block address.

Errors: If an invalid selector:offset is passed in SI:DI or not enough memory is available when increasing the block size, then this function returns carry set.

Notes: If the memory block cannot be resized in its current location, but a free block of memory of the new size exists, the memory is copied to a new block and the old one is released. The selector:offset will change if this occurs, so the SI:DI return value should be used to update all references and pointers to the memory block when this function is called.

ResMem32 Resize a previously allocated block of memory.

Inputs: AX= 0ff0eh
BX= Selector for block.
ECX= New size of block required in bytes.

Outputs: Carry set on error.

Errors: See ResMem

Notes: If the memory block cannot be resized in its current location, but a free block of memory of the new size exists, the memory is copied to a new block and the old one released. This is transparent to the application as long as only the selector originally allocated with GetMem is used to access the memory.

ResMemDOS Resize a block of DOS (conventional) memory allocated with GetMemDOS.

Inputs: AX= 0ff22h
BX= New block size in paragraphs
DX= Selector of block to modify

Outputs: Carry set on error, AX= DOS error code, BX= Maximum block size in paragraphs.

Errors: If an invalid block is passed or if not enough selectors or memory are available when expanding the block this function returns carry set.

Notes: Growing a memory block is often likely to fail since other DOS block allocations prevent increasing the size of the block. Also, if the size of a block grows past a 64KB boundary then the allocation fails if the next descriptor in the LDT is not free.

ResMemLinear Resize a previously allocated block of memory without a selector.

Inputs: AX= 0ff12h
SI:DI= Linear address of block to resize.
CX:DX= Size of block required in bytes.

Outputs: Carry set on error, else
SI:DI= New linear address of block.

Errors: If not enough memory is available when extending the block size this function returns carry set.

Notes: If the memory block cannot be expanded to the desired size, and a free block of sufficient size exists, the existing memory is copied to the free block and released. The new block is allocated in place of the old.

ResMemLinear32 Resize a previously allocated block of memory without a selector.

Inputs: AX= 0ff13h
ESI= Linear address of block to resize.
ECX= Size of block required in bytes.

Outputs: Carry set on error, else
ESI= New linear address of block.

Errors: See ResMemLinear

Notes: If the memory block cannot be expanded to the desired size, and a free block of sufficient size exists, the existing memory is copied to the free block and released. The new block is allocated in place of the old.

SetDOSTrans Set new address and size of the buffer used for DOS memory transfers.

Inputs: AX = 0ff26h
BX = Real mode segment of buffer.
DX = Protected mode selector for buffer.
ECX = Buffer size (should be <=64KB)

Outputs: None

Errors: None

Notes: The buffer must be in conventional memory and only the first 64KB will be used even if a bigger buffer is specified. CauseWay will automatically restore the previous buffer setting when the application terminates but GetDOSTrans can be used to get the current buffer's settings if you only want the change to be temporary.

You can still use the default buffer for your own purposes even after setting a new address.

SetDump Disable/enable error display and CW.ERR creation.

Inputs: AX = 0ff30h
CL = 0 if disable error display and CW.ERR file
CL = nonzero if enable error display and CW.ERR file

Outputs: None

Errors: None

Notes: By default, register dump display to screen and CW.ERR file creation are enabled on CPU faults. This option may be used to turn on and off CauseWay error processing output any number of times within an application.

SetEVect Set Protected mode exception handler address.

Inputs: AX= 0203h
BL= Exception vector number.
CX:[E]DX= selector:offset of new handler.

Outputs: None

Errors: The number in BL must be in the range 0-1Fh. Anything outside this range returns carry set.

SetMCBMax Set new memory control block (MCB) memory allocation block size.

Inputs: AX = 0ff28h
ECX = New value to set (<=64KB)

Outputs: None

Errors: Carry set on error

Notes: The maximum block size that will be allocated from MCB memory is 16 bytes less than the value set by this function. The default value is 16384. The maximum value is 65536.

The CauseWay API memory allocation functions allocate memory from two sources. Allocation requests below the value returned by this function are allocated from a memory pool controlled via conventional style MCB's. Requests above this value are allocated via the normal DPMI functions. Because DPMI memory is always allocated in multiples of pages (4KB) it can become very inefficient for any program that needs to allocate small blocks of memory. The value set by this function controls the size of memory chunks that will be allocated to and managed by the MCB system.

A value of zero can be passed to this function to disable the MCB allocation system.

The value passed will be rounded up to the nearest page (4KB) boundary.

SetRVect Set real mode interrupt handler address.

Inputs: AX= 0201h
BL= Interrupt vector number.
CX:DX= selector:offset of new handler.

Outputs: None.

Errors: None.

SetSelDet Set selector linear base and limit.

Inputs: AX= 0ff09h
BX= Selector.
CX:DX= Linear base.
SI:DI= Byte granular limit.

Outputs: Carry set on error.

Errors: If an invalid selector is passed in BX, this function returns with carry set.

SetSelDet32 Set selector linear base and limit.

Inputs: AX= 0ff0ah
BX= Selector.
EDX= Linear base.
ECX= Byte granular limit.

Outputs: Carry set on error.

Errors: If an invalid selector is passed in BX, this function returns with carry set.

SetVect Set Protected mode interrupt handler address.

Inputs: AX= 0205h
BL= Interrupt vector number.
CX:[E]DX= selector:offset of new handler.

Outputs: None.

Errors: None.

UnLockMem Unlock a region of memory.

Inputs: AX= 0ff1dh
BX:CX= Starting linear address of memory to unlock
SI:DI= Size of region to unlock in bytes

Outputs: Carry set on error.

Errors: If any of the memory region specified is invalid this function returns carry set.

Notes: This function allows the unlocked memory to be swapped to disk by the VMM if necessary. Areas below and above the specified memory to the nearest page (4KB) boundary are unlocked if the specified region is not aligned to a page boundary.

UnLockMem32 Unlock a region of memory.

Inputs: AX= 0ff1eh
ESI= Starting linear address of memory to unlock
ECX= Size of region to unlock in bytes

Outputs: Carry set on error.

Errors: See UnLockMem

Notes: This function allows the memory to be swapped to disk by the VMM if necessary. Areas below and above the specified memory to the nearest page (4KB) boundary are unlocked if the specified region is not aligned to a page boundary.

UserDump Setup user-defined error buffer dump in CW.ERR.

Inputs: AX = 0ff2fh
ES:[E]DI - user buffer to display in CW.ERR
CX = count of bytes to display from buffer in CW.ERR
BL = 'A' if ASCII dump (non-binary display of bytes, control characters display as periods)
BH = nonzero if preset ASCII buffer to word value, ignored for non-ASCII
DX = word value to fill ASCII dump buffer if BH is nonzero, ignored for non-ASCII

Outputs: Carry set on ASCII dump invalid buffer address.

Errors: The user buffer must be a valid readable selector and offset value when this function is called or else the request is ignored and a carry flag condition is returned. If BH is set to nonzero to flag presetting the buffer bytes, the selector must be writable. Specifying a larger CX count than available buffer size will also return an error.

Notes: If the fill ASCII buffer condition is specified, any values previously in the buffer will be overwritten when this call is made.

UserErrTerm Call user error termination routine.

Inputs: AX = 0ff31h
CL = 0 if 16-bit termination routine
CL = nonzero if 32-bit termination routine
DS:[E]SI = user termination routine address. If DS is zero, the user termination routine call is removed.
ES:[E]DI = Information dump buffer address, 104 bytes. If ES is zero, no information dump is performed.

Outputs: None

Errors: None

Notes: The user termination routine is responsible for returning to the CauseWay termination routines to allow proper shutdown of the application. The instruction must be the proper 16- or 32-bit return to match the CL register setting. For ease of use with high-level languages (specifically Watcom C and setting SS back to DGROUP), [E]SI equals the internal DOS extender stack ESP immediately prior to the 32- or 16-bit call to the termination routine. If an information dump buffer address is provided, register and other termination values are placed into it using the following format:

```
dword           EBP;
dword           EDI;
dword           ESI;
dword           EDX;
dword           ECX;
dword           EBX;
dword           EAX;
word            GS;
word            FS;
word            ES;
word            DS;
dword           EIP;
word            CS;
word            reserved1;
dword           EFLAGS;
dword           ESP;
word            SS;
word            reserved2;
word            TR;
dword           CR0;
dword           CR1;
dword           CR2;
dword           CR3;
dword           csAddress;
dword           dsAddress;
dword           esAddress;
dword           fsAddress;
dword           gsAddress;
dword           ssAddress;
word            ExceptionNumber;
dword           ErrorCode;
```

7.6 API Notes

A fixed segment selector at 40h is always available to the application. This selector maps the real mode memory at 400h where most of the BIOS variables can be found. CauseWay also provides selectors at standard video addresses 0B000h, 0B800h and 0A000h in non-DPMI environments to ease conversion of real mode code.

The environment variable block address in the Program Segment Prefix (PSP) is a valid protected mode selector. A valid protected mode selector:offset is also placed in the PSP at offset 34h for the file handle list pointer. Note that the default value in the program's PSP will point to the real mode PSP, not the protected mode PSP, even if the handle count is less than or equal to twenty. Code that makes use of the handle list should use the address at PSP+34h rather than assuming the list's position within the PSP. Also, when an application is operating non-DPMI conditions the handle table will have an entry for CauseWay's VMM swap file.

DOS functions which use the obsolete file control blocks (FCBs) are not supported by CauseWay, although such support may be added by the developer.

CauseWay applications should terminate using INT 21H function 4Ch. As with real mode operation, the error level passed to this function in the AL register is returned to the parent program or DOS.

If a CauseWay application needs to terminate and stay resident (TSR), then INT 21H function 31h may be used. Unlike real mode operation, no memory value is required for this function. All memory owned by the application when the TSR function is issued remains the program's property. There is currently no way of removing the CauseWay application from memory once it becomes a TSR without rebooting the machine or using a third party TSR manager. However, a TSR manager will not automatically release extended memory allocated for the CauseWay TSR.

Unhandled exceptions terminate the program with a register display dump to screen and a text file called CW.ERR. CW.ERR contains other potentially useful information about the state of the application when it terminated. Refer to the appendices for more information on the CW.ERR file information format.

CauseWay runs applications at privilege level 3. Privilege level 0 reserved instructions will cause a general protection fault (GPF). CauseWay emulates the four instructions MOV EAX,CR0; MOV CR0,EAX; MOV EAX,CR3; and MOV CR3,EAX in the GPF handler so that they may be used by an application.

CauseWay will match a DPMI 0A00h function call with target string RATIONAL/4G. This in conjunction with CR0 emulation allows support of floating point emulation by an exception handler. Watcom uses this approach under DOS/4GW operation if the floating point emulation library is not linked in. Note that this routine does not work for either CauseWay or DOS/4GW under a DPMI host such as Windows or OS/2.

8 Interrupt Services

8.1 Extended or Altered Interrupt Services

The size of registers used by CauseWay's extended or modified interrupt services depends on the limit of the selector. Extended 32-bit registers are used for 32-bit sized selectors and 16-bit registers are used for selectors within 16-bits. To reflect this, the convention of an [E] in brackets is used before a listed register when it must be a 32-bit value with a 32-bit selector and a 16-bit value with a 16-bit selector.

Required registers that are not specified in this list should be set up in the same way as required for normal DOS real mode operation. For INT APIs that are not listed and require segemnt pointers, either handle them using the CauseWay IntXX function or create your own interrupt translation code.

INT 10h

10h sub function 02h, [E]DX instead of DX.
10h sub function 09h, [E]DX instead of DX.
10h sub function 12h, [E]DX instead of DX.
10h sub function 17h, [E]DX instead of DX.
13h [E]BP instead of BP.
1Ch sub function 01h, [E]BX instead of BX.
1Ch sub function 02h, [E]BX instead of BX.

INT 21h

09h [E]DX instead of DX
0Ah [E]DX instead of DX
0FH - 17H not supported; use corresponding file handle function.
1Ah [E]DX instead of DX
21h - 24h not supported; use corresponding file handle function.
25h [E]DX instead of DX. Protected mode vector will be set.
26h - 29h not supported; use corresponding file handle function.
2Fh [E]BX instead of BX
31h No value is required in DX
35h [E]DX instead of BX. Protected mode vector will be returned.
39h [E]DX instead of DX
3Ah [E]DX instead of DX
3Bh [E]DX instead of DX
3Ch [E]DX instead of DX
3Dh [E]DX instead of DX
3Fh [E]DX instead of DX
40h [E]DX instead of DX
41h [E]DX instead of DX
43h [E]DX instead of DX
44h subfunction 02h, use [E]DX instead
44h subfunction 03h, use [E]DX instead
44h subfunction 04h, use [E]DX instead
44h subfunction 05h, use [E]DX instead

47h [E]SI instead of SI
48h Protected mode memory will be allocated
49h Protected mode memory will be released
4Ah Protected mode memory will be resized
4Bh [E]DX & [E]BX instead of DX & BX
Parameter block offset entries are [d]word
4Eh [E]DX instead of DX
56h [E]DX & [E]DI instead of DX & DI
5Ah [E]DX instead of DX
5Bh [E]DX instead of DX
62h Protected mode selector will be returned
6Ch [E]SI instead of SI

INT 23h Control-C Handler Address

This interrupt is always reflected back to the protected mode handler to ensure the CauseWay application can handle it correctly. The default handler aborts the application in the same manner as DOS. If you need to terminate your application in your own handler, perform an INT 21h AH=4ch as normal.

INT 24h Critical Error Handler Address

This interrupt is always reflected back to the protected mode handler to ensure the CauseWay application can handle it correctly. The default handler behaves in the same way as the DOS handler and it aborts your application, if appropriate. If you install your own handler, all memory accessed by this interrupt as code or data must be locked.

The register values normally placed on the stack by DOS before entry to the interrupt handler are not present in protected mode. Only the register values are valid. You may terminate your application from within this interrupt with INT 21h, AH=4ch as normal.

INT 33h

09h [E]DX instead of DX
0Ch [E]DX instead of DX
16h [E]DX instead of DX
17h [E]DX instead of DX

Notes:

With the exception of software interrupts that require segment pointers as parameters, all interrupts can be issued as normal. The most common interrupt APIs that require segment pointers are intercepted by CauseWay to provide normal access to these services. Any other real mode interrupt services that require segment pointers can be accessed using CauseWay's simulated real mode interrupt/far call services.

Hardware interrupts are always reflected to protected mode handlers even when signaled during real mode operations. This ensures that protected mode applications always retain control without requiring you to patch real mode interrupt vectors. The remaining interrupts are serviced via the vector table appropriate to the mode. Use the real to protected mode callback services to provide real mode code with access to protected mode code, and allow any interrupt to be re-signaled in protected mode.

If you add your own hardware interrupt handlers, suchas the timer tick at vector 08h, any memory that the handler reads or writes, including its code, must reside in locked memory. (CauseWay provides a locked stack.) This limitation is required because DOS is not re-entrant and hardware interrupts can occur at any

time. Interrupts occurring during DOS activity prevent CauseWay's virtual memory manager from accessing its swap file. Lock memory will not move to the swap file.

9 Troubleshooting

9.1 First Steps

If you have problems using CauseWay, first try linking and running a one-line program that simply prints "Hello" on your computer screen. This will help establish if the problem is a basic incompatibility with CauseWay and your system setup, or if the error may lie elsewhere (e.g. a third party library).

The remainder of this chapter provides a description of error and warning messages that you may encounter when using CauseWay. Suggested solutions to correct the errors are included where possible.

9.2 DOS Extender Error Messages and Return Values

DOS extender error messages are displayed by the CauseWay DOS extender when a CauseWay application is running and encounters a serious problem that it cannot recover from. The DOS extender then terminates the application with the appropriate return code, displaying a dump of register values, and writing system information to the file CW.ERR.

Error 01 **Unable to resize program memory block.**

Generated if DOS reports an error when CauseWay tries to resize its real mode memory block. As the block is always shrunk, the only possible cause of this is corrupted memory control blocks (MCBs). Reboot the system to correct this error.

Error 02 **386 or better required.**

Generated if CauseWay is run on any machine with a processor below a 386SX. To correct this error, run the application on another machine or upgrade the machine's processor.

Error 03 **Non-standard protected mode program already active.**

Generated if the system is already operating under the control of another protected mode program which doesn't conform to either VCPI or DPMI standards. Identify and remove the other application before running the CauseWay application.

Error 04 **DOS 3.1 or better required.**

Generated if DOS version is less than 3.1. You need to upgrade the machine's DOS version or use another machine to operate the CauseWay application.

Error 05 **Not enough memory for CauseWay.**

Generated if the system doesn't have enough free physical memory to initialize the CauseWay kernel code and data. Free additional memory before running the CauseWay application. The memory can be any of the extended or conventional types supported by CauseWay.

Error 06 **VCPI failed to switch into protected mode.**

Generated if a VCPI server is detected and the server fails to switch into protected mode when requested. The only likely cause of this error is a corrupted system. Reboot the system and try again.

Error 07 **Unable to control A20.**

Generated if CauseWay detects an A20 line that doesn't respond to the normal control methods. This may indicate either a hardware fault or a nonstandard system. There is no software solution for these hardware problems. Installing an XMS driver such as HIMEM.SYS should address nonstandard systems.

Error 08 **Selector allocation error.**

Generated if DPMI refuses to allocate enough selectors for CauseWay to function. Remove one or more programs that are also using DPMI.

Error 09 **Unrecoverable exception. Program terminated.**

This is the standard General Protection Fault, or GPF, message. It is generated if a nonrecoverable exception occurs which suggests a bug in the application. Use the register dump displayed with this message along with the information in CW.ERR and the program's .MAP file to help track down the location and cause of the problem.

Error 10 **Unable to find application to load.**

Generated if CauseWay is unable to find the application within the executable .EXE file. This situation indicates a corrupted file. Rebuild or obtain another copy of the application.

Error 11 **DOS reported error while accessing application.**

Generated if any kind of error is detected while accessing the CauseWay application executable file. This situation indicates a corrupted or missing file. Rebuild or obtain another copy of the application.

Error 12 **Not enough memory to load application.**

Generated if CauseWay is unable to provide enough memory to load the application. Free additional memory and/or disk space before running the application. Check for CAUSEWAY=SWAP, TEMP and TMP environment variables that point to a disk with little free space. If running under an operating system that provides DPMI per application, increase the application's DPMI allocation.

Error 13 **DPMI failed to switch to protected mode.**

Generated if the machine is using a DPMI server and it fails to switch to protected mode. If the DPMI server only supports multiple clients of the same type (either 16- or 32-bit) then the problem is probably that different types of applications are already being run. Remove the other type of DPMI application(s) before running the CauseWay application.

Error 14 **Memory structures destroyed. Program terminated.**

Generated if internal memory management structures become corrupted. This is caused by the CauseWay application writing to memory regions that have not been allocated to it and is a bug in the application. Obtain a corrected version of the application to fix this error.

Error 15 **DOS reported an error while accessing swap file. Program terminated.**

Generated if any level of error is detected while accessing the swap file. The swap file has probably been deleted inadvertently by the application or perhaps marked as read-only.

Error 16 **Unsupported DOS function call. Program terminated.**

The CauseWay application attempted to use an obsolete DOS function which used file control blocks (FCBs). Use the file handle DOS functions in the application instead.

B

C

D

E

F

H

I

K

L

M

N

P

R

S

T

V

W