Developer's Guide

Validator for

OS/2 *Version 1.0*

Advanced 32-Bit OS/2® Application Development Support

Version 1.0 Draft 11

22-Apr-1994 5:00:22 PM

Prominare Inc.
100 Front Street East
Toronto, Ontario
Canada
M5A 1E1

Document: 19940402-011-100-DG

Table of Contents

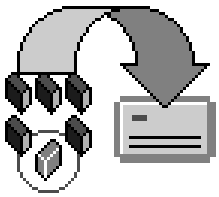
List of Plates, Figures and Tables

Plates

Figures

Tables

Installation



Validator is a full functioning, professional programming analysis tool specifically designed for your needs; that of a professional programmer using the OS/2 2.x environment. ***Validator*** has been designed for speed and ease of use in helping you locate problems within your applications without having to utilize a debugger or writing specilized code. ***Validator*** provides additional information on errors or possible errors when you use the 32-bit OS/2 API's.

Validator allows you to monitor, through its ***ViewPort*** tool, error return codes along with parameter violations. This allows you to run your applications as you would normally while benefiting from the ability of knowing when and where an OS/2 API failed with the exact error code.

You will find that by using ***Validator***, your approach to building your application is greatly simplified since you can receive exact information why something failed including the line number within the source file. Situations where you may have spent long hours tracking down a problem can now be solved in a matter of minutes especially if that problem involved an OS/2 API.

To install the ***Validator*** onto the hard disk of your computer, place the first package diskette in Drive A:, start an OS/2 command prompt session and type the following:

a: ↵ENTER

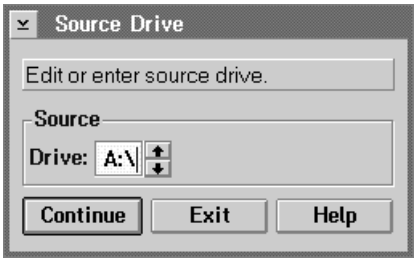
install ↵ENTER

The installation program will first display the welcoming panel, as shown in Plate 1.1.

Plate 1.1
Welcome panel

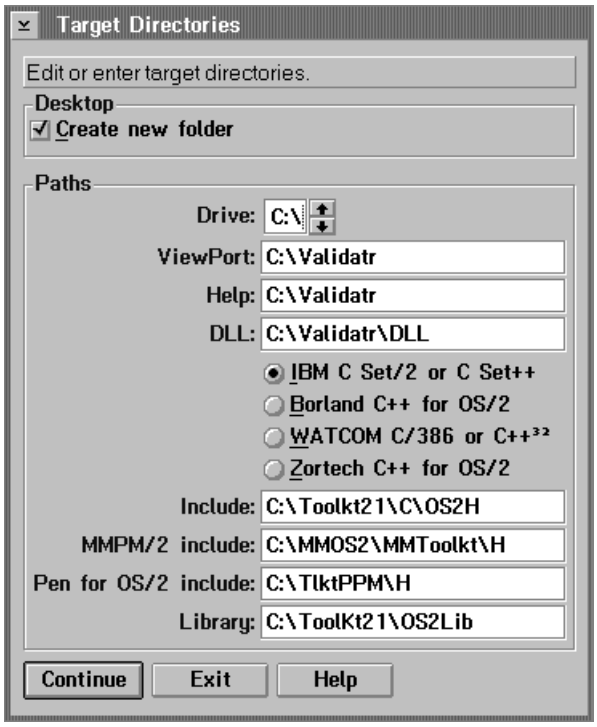
It will then display a dialogue, as shown in Plate 1.2, for the source drive that will be used to read the program files from. You can change the default drive to another diskette drive if you desire.

Plate 1.2
Source Drive dialogue



Again, you can press the ENTER key or click the mouse pointer on the **Continue** push button. This will cause the **Target Directories** dialogue, Plate 1.3, to be displayed. It is through this dialogue that you indicate where you want to place the ***Validator*** files and will automatically provide default locations. You can change them if you want to place the files into different directories or on another drive on your hard disk. The locations entered will be recorded so that the next time you update the applications, the directories entered will be automatically selected.

Plate 1.3
Target Directories dialogue



The entry fields are labeled as:

Table 1.1
Installation paths

Entry Field	Purpose
ViewPort	Location where the ***ViewPort*** application files will be placed. Default: C:\Validatr
Help	Location where help files will be placed. Default: C:\Validatr\Help
DLL	Location where ***Validator*** .DLL files will be placed. Default: C:\Validatr\DLL
Include	Location where include header files will be placed. This should be the location where your OS/2 toolkit include files are located. Default: C:\Toolkt21\C\OS2H

Entry Field	Purpose
MMPM/2 include	Location where include header files will be placed utilized by the MMPM/2 toolkit. This should be the location where the MMPM/2 include files are located. Default: C:\MMOS2\MMToolkt\H
Pen for OS/2 include	Location where include header files will be placed utilized by the Pen for OS/2 toolkit. This should be the location where the Pen for OS/2 include files are located. Default: C:\TlktPPM\H
Library	Location where the import libraries for ***Validator*** are to be placed. Default: C:\Toolkt21\OS2Lib

You can from this dialogue also instruct the installation program to create a program folder that will contain ***ViewPort*** and the example applications. By selecting the **Create new folder** option, the program folder will be created.

N O T E

You do not require MMPM/2 or Pen for OS/2 to be able to use ***Validator***.

Validator has been designed to work on OS/2 2.0, OS/2 2.1, OS/2 2.11 and with or without MMPM/2 or Pen for OS/2. If you are not using MMPM/2 or Pen for OS/2, you should enter the current OS/2 Toolkit header location for either of the header areas.

Also, you need specify the compiler you are using to allow the proper .DLL's to be installed. There are slight differences in how the different compilers comply with the system API calling conventions which dictate different versions of the validation .DLL's. These differences will be explained in Appendix B (see page 102). The following options are provided:

Table 1.2
Compiler selections

Radio Button	Compiler
IBM C Set/2 or C Set++	IBM C Set/2 Version 1.0 or IBM C Set++ Version 2.0.
Borland C++ for OS/2	Borland C++ for OS/2 Version 1.0 or 1.5
WATCOM C/386 or C++32	WATCOM C/386 Version 9.0 or WATCOM C/C++32 Version 9.5.
Zortech C++ for OS/2	Zortech C++ for OS/2 Version 3.1.

Plate 1.4
Installation Progress window

Once you have finished entering or changing the destinations for the ***Validator*** files and have selected the target compiler, you can click on the **Continue** push button or press the ENTER key. This will start the process of installing the files contained on the diskette. The installation progress window, Plate 1.4, will display the source and destination of the files that are being copied.

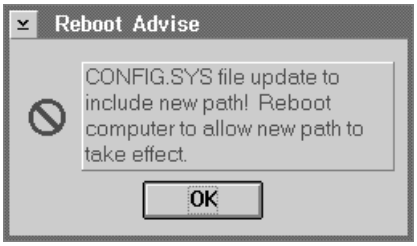
If a path cannot completely fit within the displayed area, it will show an abbreviated form of the full path where it will substitute ... for some path component or components. It will always try to display the filename that is part of the path.

The progress of the installation will be displayed through a status indicator which shows the percent complete. When all of the files have been copied to your system from the diskettes, the installation program will check your CONFIG.SYS file to see if needs to be updated for the locations where you placed the files for the ***Validator***. If the locations within the PATH, LIBPATH, DPATH or HELP are not correct, the **Update CONFIG.SYS** dialogue as shown in Plate 1.5 will be displayed.

Plate 1.5
CONFIG.SYS Update dialogue

The dialogue allows you to specify how the configuration information should be updated. The installation program can update the CONFIG.SYS file to ensure that the required locations are correct so you can properly use the ***Validator***. For this to occur, you need to select the **CONFIG.SYS file** option in the **Save changes to** group. When you select the **File named CONFIG.NEW**, you will have transfer the changes from this file to CONFIG.SYS manually otherwise ***Validator*** may not operate correctly. A good indication of this is that the ***ViewPort*** tool will start.

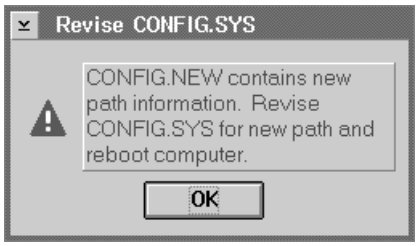
Plate 1.6
Reboot dialogue



By having the installation program update the CONFIG.SYS file, a dialogue will be displayed reminding you to reboot your machine before starting any of the applications installed for the ***Validator***. This is required due to the DLL's that are needed by the applications.

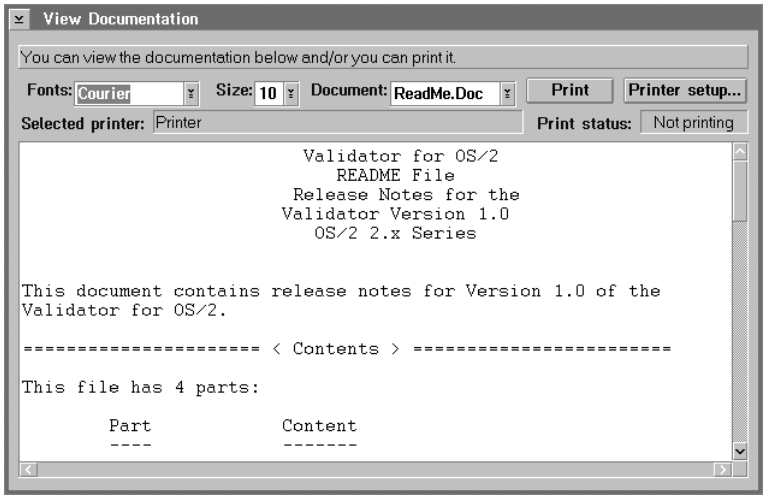
This dialogue, as shown in Plate 1.6, is just informational. It will not be displayed if you choose to place the updated configuration information within the CONFIG.NEW file. In place of the **Reboot Advice** dialogue will be a dialogue, as shown in Plate 1.7, advising you to update your CONFIG.SYS file. If you do not revise your CONFIG.SYS file, you may not be able to use ***Validator*** since it will not be able to find required files, specifically the DLL's files.

Plate 1.7
Revise CONFIG.SYS dialogue



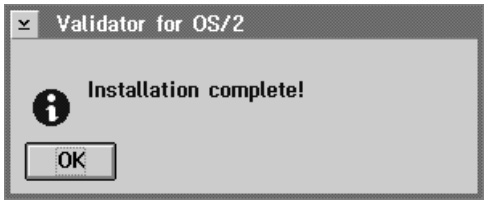
Before exiting, a dialogue will be displayed, as shown in Plate 1.8, allowing you to read the on-line installation documentation which includes the ReadMe.Doc and Packing.Lst files.

Plate 1.8
View Documentation dialogue



Once the ***Validator*** has been successfully installed, one last dialogue will be displayed, as shown in Plate 1.9, stating that installation process is complete.

Plate 1.9
Installation Complete Dialogue



How to Begin

Since the ***Validator*** applications have been designed for ease of use, you can immediately begin to use them upon installation. Before beginning to use the ***Validator*** full time, you should read Sections 2, **Overview** (see page 9) and Section 3, **API's** (see page 27) before trying to utilize features of ***ViewPort***.

Validator *Design*

All of the applications within the ***Validator*** adhere to IBM's *System Application Architecture* **Common User Access Guide to User Interface Design** and **Common User Access Advanced Interface Design Reference.**

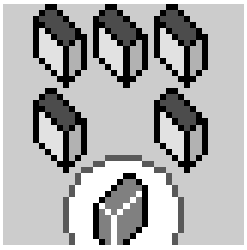
Notational Conventions

This manual uses the following notational conventions in defining program usage, revisions, references, etc. and in presenting examples:

Table 1.3 Conventions

Convention	Meaning
Bold type	Item of special interest, menu item, or program name.
[ ]	Option input values
drive:	Drive designation
Esc	Escape Key
↵ENTER	Enter/return key
OS/2	**IBM OS/2** Operating System/2
Operating System	**OS/2** unless specifically stated
DLL	Dynamic-Link Library
dirname	Directory name
filename	Filename
control	Control type such as push button, button, action bar menu or sub-menu
⇓	Input through parameter.
button 1	Refers to button 1 on the mouse. Generally, the left mouse button is button 1 unless the buttons have been swapped through the control panel.
button 2	Refers to button 2 on the mouse. Generally, the right mouse button is button 2 unless the buttons have been swapped through the control panel.
shred	Equivalent to delete or erase.
`text`	Keyboard input or program display output

Overview



What is ***Validator***? It is a set of DLL's and executables that work in conjunction with the OS/2 API's providing additional and more detailed error information regarding your usage of a given 32-bit OS/2 API. The basic premise of ***Validator*** is to provide details indicating which parameter you have passed to an OS/2 API that may be incorrect. The result indicates exactly which parameter is in error. The OS/2 API's will let you know that you have an invalid parameter but due to system design where responsiveness is a high priority, each API cannot provide the detail as to which parameter in the API is in error. Also, the API's use fast determination of errant flag values, again, for responsiveness.

This can be seen with the **DosOpen** API which is defined as:

Figure 2.1
DosOpen API

```
APIRET DosOpen( PSZ pszFileName, PHFILE pHf, PULONG pulAction,
                ULONG cbFile, ULONG ulAttribute, ULONG fsOpenFlags,
                ULONG fsOpenMode, PEAOP2 peaop2 );
```

where there are reserved values defined for the open mode flags, *fsOpenMode*. The API to validate these flags would use a value of 0xffff0c04 to isolate the reserved bits by AND'ing the value with the *fsOpenMode* parameter value and if any of the resultant bits were set, it would return an error of ERROR_INVALID_PARAMETER. Unfortunately, you can't easily determine which parameter is in error through the return code.

Validator *methodology*

Validator solves this problem by masking out the reserved bits individually and depending on which set of reserved bits has been set, it will return a specific error code indicating which reserved bits are set and in what parameter. Using the **DosOpen** again as the example, if bit 11 of *fsOpenMode*, which is a reserved bit, was set, the return code returned by the **DosOpen** API would be a combination of ERROR_INVALID_PARAMETER OR'ed with PERR_DO07_INVALIDOPENMODEBITS11 (0x001a0057). The 07 of the PERR_DO07 indicates that it was parameter 7, or the *fsOpenMode* parameter. Each OS/2 API parametter error defined for ***Validator*** operates in this manner. It is designed to allow you to quickly determine the parameter in questions.

Error code construction

To be able to better understand how the error codes are formed denoting a parameter error, you first need to understand the current error forms. For Dos*, Prt*, some Spl*, Pen for OS/2 and MMPM/2 calls the normal error form is:

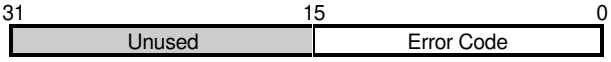


Figure 2.2 Error code layout Dos*, Prt*, some Spl*, Pen for OS/2 and MMPM/2 calls

and for the Ddf*, Dev*, Drg*, Gpi*, Pic*, Prf*, some Spl* and Win* calls, the normal error form is:

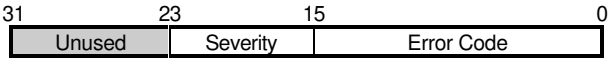


Figure 2.3 Error code layout Ddf*, Dev*, Drg*, Gpi*, Pic*, Prf*, some Spl* and Win* calls

Validator makes use of the unused portion of the error codes to provide the additional error information. In the case of the Dos*, Prt*, some Spl*, Pen for OS/2 and MMPM/2 calls the parameter error form is:

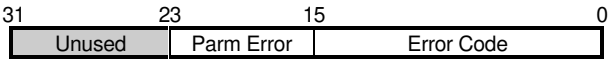


Figure 2.4 Adapted error code layout Dos*, Prt*, some Spl*, Pen for OS/2 and MMPM/2 calls

and for the Ddf*, Dev*, Drg*, Gpi*, Pic*, Prf*, some Spl* and Win* calls, the parameter error form is:

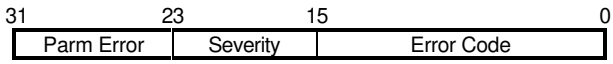


Figure 2.5 Adapted error code layout Ddf*, Dev*, Drg*, Gpi*, Pic*, some Spl* and Win* calls

Therefore, to properly interpret the error codes returned to your application, if the bytes where the parameter error would be stored is zero (0), then the error code being returned is an actual error code from the API used. When the value of the bytes where the parameter error is stored is non-zero (>0), then the call was failed due to a problem detected within the validation routines with one of the parameters. You will need to view the return code in hexadecimal when using a debugger such that you can isolate the bytes where the error information would be returned. Then, using the *Parameter Error Reference*, you can determine the problem by locating the function within the reference and then matching the parameter error with the values listed within the table corresponding table for the API.

If you are utilizing the ***ViewPort*** facility, then you can let ***ViewPort*** do the hard work for you which will decode the information as well as provide additional information such as the index within an array that contained the failing element.

OS/2 - Validator architecture

So, really how does ***Validator*** work? Figure 2.6 depicts the standard architecture of an OS/2 application working in conjunction with the system. You will notice that the application makes an API call which is then handled by the appropriate DLL which then may repackage the call to the kernel or handle it directly.

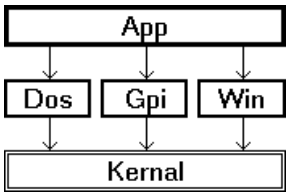


Figure 2.6 OS/2 application architecture

When you allow for ***Validator*** usage with an application, the architecture is changed slightly, as shown in Figure 2.7.

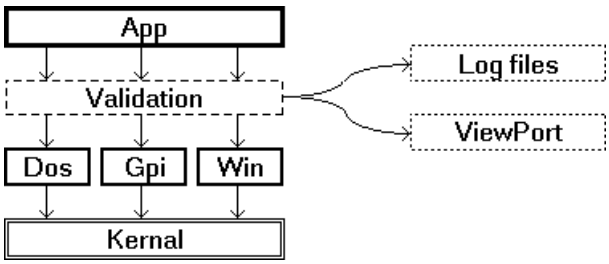


Figure 2.7 OS/2 application architecture with validation support

Between your application and the system DLL's are the validation support DLL's. If an error is detected through the validation DLL, the appropriate return code is sent back to the application. If additional levels of support are to be provided through the DLL's, then other factors come into play. If ***ViewPort*** support has been requested, then the error information is sent to it by the validation routines even if the error was a normal API error. Also, if error logging was requested, the error information would be written to a specified file in ASCII format or a format that can be viewed through ***ViewPort***.

Looking at an API in detail can make it easier to understand what is happening. In Figure 2.8, the **DosCreateDir** API is being used to create a new directory. The values of the call would be passed directly to the entry point for the call within the DOSCALLS DLL. It would in turn then pass the call to the operating system kernel for servicing.

Figure 2.8 **DosCreateDir** calling sequence

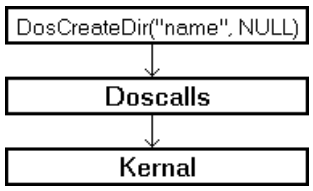


Figure 2.9 shows how the validation process changes this mechanism. The same call is used but in this case it is first passed to the entry point within the validation DLL's. If the first parameter, which is for the directory name is a valid pointer and the second parameter, which is for the extended attributes of the directory is either NULL or a valid pointer, the call is then passed to the entry point within the DOSCALLS DLL. It would then pass the request to the kernel for servicing. If the kernel detects an error condition, it returns an error code back as the return value. The validation routines would then take the value and if error logging was requested, the error information would be sent to the logging file or if ***ViewPort*** support for the application was requested, the error information would be sent to ***ViewPort***. The error code would then be returned back to the application which issued the **DosCreateDir** call.

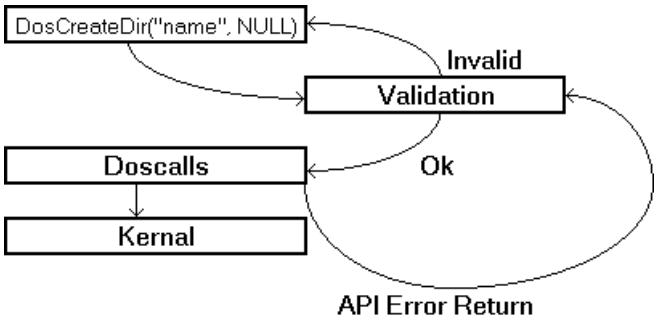


Figure 2.9 **DosCreateDir** calling sequence with validation support

If the directory name or extended attribute parameters were determined to be an invalid pointer, the validation routine would construct the appropriate error return code indicating which parameter was in error and for what reason. This return code would then be passed back to the application that made the **DosCreateDir** call.

You will notice that this description of how the call works when a parameter error is detected did not mention that the actual **DosCreateDir** entry point is called. Well, since an error code would be returned anyways and there is no good reason to tie up the system indicating basically the same problem.

In the case of API's that rely on the **WinGetLastError** mechanism, the sequence is similar except for the fact that an error code is not returned back to the application but the value (such as a FALSE to indicate failure) that is defined to indicate an error condition.

Figure 2.10 shows essentially the same calling sequencing as does Figure 2.8. The **WinInitialize** API is being used to initialize the PM facilities for an application. The value of the call would be passed directly to the entry point for the call within the PMWIN DLL. It would in turn then pass the call to the operating system kernel for servicing if necessary.

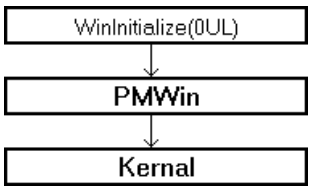


Figure 2.10 **WinInitialize** calling sequence

Figure 2.11 shows how the validation process changes this mechanism. The same call is used but in this case it is first passed to the entry point within the validation DLL's. If the parameter for the initialization option is valid, the call is then passed to the entry point within the

PMWIN DLL. It would then pass the request off to the kernel for servicing if necessary. If an error condition is detected, a zero return value is used instead of the anchor block handle which would be the successful return value.

The validation routines would then take the value and if error logging was requested, the error information would be determined and then sent to the logging file or if ***ViewPort*** support for the application was requested, the error information would be sent to ***ViewPort***. The return value would then be returned back to the application which issued the **WinInitialize** call. If the application was interested in the error value, it would then have to issue a **WinGetLastError** call to retrieve the error value from PM.

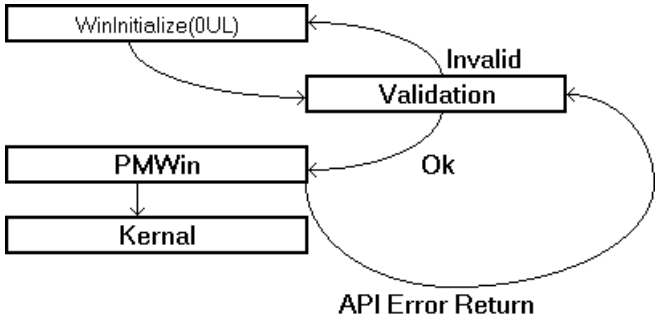


Figure 2.11 **WinInitialize** calling sequence with validation support

If the parameter was not zero (0), which is the only valid value for the parameter, the validation routine would construct the appropriate error return code indicating which parameter was in error and for what reason and inform PM of the error condition. This return value of zero (0) would then be passed back to the application that made the **WinInitialize** call. The application on receiving the failure indication could then issue the **WinGetLastError** call to determine the problem. It could then at this point see if it was a parameter error by checking the high byte of the error code.

You may have noticed that this description of how the call works operates in a similar manner to that of the **DosCreateDir** in that when a parameter error is detected that the actual **WinInitialize** entry point is not called.

*Using **Validator***

Understanding how the validation routines do their job will allow you to better understand how you can utilize the features of ***Validator***. At its most basic level, when you link an application with the VALIDATR.LIB import library in place of the OS2386.LIB, you allow the additional parameter return codes to be received by your application.

This may be useful when debugging your application since the return codes can be viewed through the debugger usually within a return code variable or the **EAX** register. It should be noted that this is only true with Dos*, Prt*, some Spl*, Pen for OS/2 and MMPM/2 calls. In the case of the Ddf*, Dev*, Drg*, Gpi*, Pic*, some Spl* and Win* calls, you will not be able to see the error information unless the debugger that you are using can do a **WinGetLastError** on behalf of your application.

Logging file support

The next level of support, which essentially builds on this base feature, allows you to log the error information to a log file. To allow this to happen, one of the API's for ***Validator*** is used. **ValInitialize** (see page 55) has the following format:

Figure 2.12 **ValInitialize** definition

HVAL ValInitialize(*PSZ pszAppName, PSZ pszLogFile, ULONG ulSupport* **);**

It is used to initialize the validation routines for the additional support based on the support level requested through the *ulSupport* parameter. The log file filename is defined through the *pszLogFile* parameter of the call. Depending on the flags set within the *ulSupport* parameter, you can specify that the logging file is to be in ASCII format (VL_ERRORLOG), like that shown in Figure 2.13, or to be saved in a format that ***ViewPort*** can read in for detailed viewing. You can also through the *ulSupport* parameter reset the logging file such that it is deleted with an option to insure that the deleted file is not saved to the undelete directory if the user has allowed the undelete feature of OS/2 to be active.

Figure 2.13 ASCII error log data

```
Jul-07-1993 07:52:55.94
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
           Error: 0x00000012  (18)
      Process ID: 48
       Thread ID: 1
Jul-07-1993 07:52:56.13
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
           Error: 0x00000012  (18)
      Process ID: 48
       Thread ID: 1
Jul-07-1993 07:52:56.19
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
           Error: 0x00000012  (18)
      Process ID: 48
       Thread ID: 1
Jul-07-1993 07:52:56.32
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
           Error: 0x00000012  (18)
      Process ID: 48
       Thread ID: 1
```

Another level of support provided, is where the source file filename and line number where the API was invoked from is recorded. When you are

using log file support in ASCII, the file and line number that the API was called from is also shown with the information recorded like that within Figure 2.14 below:

Figure 2.14
ASCII error log data

```
Jul-07-1993 08:06:33.66
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
            File: Dialogs.C
            Line: 99
 Error: 0x00000012  (18)
      Process ID: 48
       Thread ID: 1
Jul-07-1993 08:06:34.10
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
            File: Dialogs.C
            Line: 145
           Error: 0x00000012  (18)
      Process ID: 84
       Thread ID: 1
Jul-07-1993 08:06:34.16
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
            File: Dialogs.C
            Line: 99
           Error: 0x00000012  (18)
      Process ID: 84
       Thread ID: 1
Jul-07-1993 08:06:34.38
     Application: DirList.Exe
     Failing API: DosFindNext
       Parameter: N/A
            File: Dialogs.C
            Line: 145
           Error: 0x00000012  (18)
      Process ID: 84
       Thread ID: 1
```

The other form of logging file is in a binary format that is readable by ***ViewPort***. The information recorded is in greater detail such that when the information is loaded by ***ViewPort*** you can treat the information as though ***ViewPort*** had been receiving it in real time.

These logging files are useful in that you can provide this support within your applications while you are testing even with third parties. Although the speed of your application will be up to 5 to 10% slower, depending on the calls you are using, the information recorded regarding the OS/2 API's that detected errors will greatly aid you in solving problems that would be very difficult to try to determine exactly what may be going wrong at the third party site.

You can design the test software to have different levels of support that are activated when you provide a command line switch. For example, you may want the logging file to be reset each time the user starts the application but if problems are encountered within the application, you may wish the user to activate a command line switch that causes the logging information to be cumulative. Figure 2.15 shows an example of this.

Figure 2.15 Dynamic initialization

```
INT main(INT argc, CHAR *argv[ ])

{
HVAL  hval;                         /* Validator Handle                         */
ULONG ulSupport = VL_ERRORLOG;      /* Validation Support Level                 */

if ( argc == 2 )
     if ( !stricmp(argv[1], "-C") && !stricmp(argv[1], "/C") )
          ulSupport |= VL_LOGRESET;

hval = ValInitialize("Example.Exe", "Example.Log", ulSupport);
```

The most useful level of the logging information in the ASCII format is when you allow the source file filename and line number to be recorded. This will help you to localize the problem to the file and line within your source code. Again, this can be handled through a command line switch like that shown below:

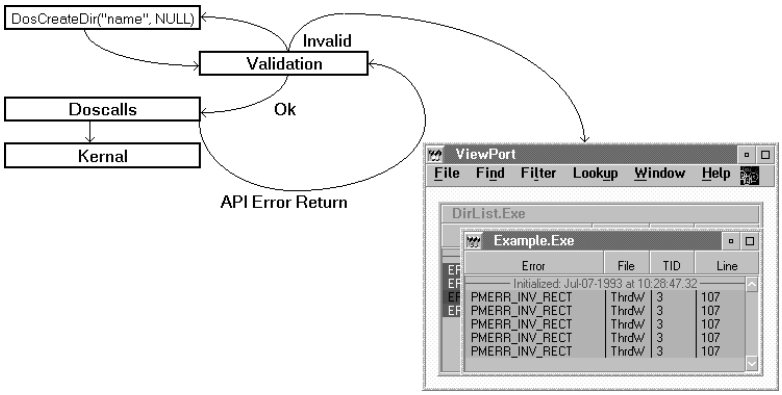
Figure 2.16 Dynamic initialization

```
INT main(INT argc, CHAR *argv[ ])

{
HVAL  hval;                         /* Validator Handle                         */
ULONG ulSupport = VL_ERRORLOG;      /* Validation Support Level                 */

if ( argc == 2 )
     if ( !stricmp(argv[1], "-F") || !stricmp(argv[1], "/F") )
          ulSupport |= VL_FILELINE;

hval = ValInitialize("Example.Exe", "Example.Log", ulSupport);
```

By designing your support level during the test phase, you can provide a level of support where your remote users can provide feedback almost immediately without having to transmit a file back to your location as would be the case if the logging file was in ***ViewPort*** format.

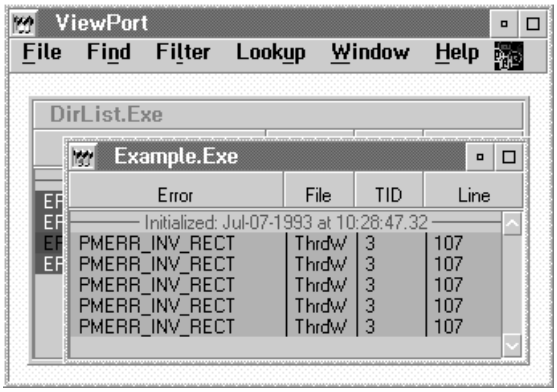
For in-house usage, you will probably want to utilize ***ViewPort*** to monitor the activity of OS/2 API and parameter errors within the application. ***ViewPort*** works in conjunction with the validation DLL's where the validation DLL's send information to ***ViewPort*** when an OS/2 API or parameter error is detected. Figure 2.17 depicts the mechanism:

Figure 2.17 **DosCreateDir** calling sequence with validation and ***ViewPort*** support



When an OS/2 API error or a parameter validation error is detected, ***Validator*** will, along with the return code sent back to the application that issued the API call, send the information to ***ViewPort*** at the same time. This will allow you to be able to see within a ***ViewPort*** error window the error information recorded. The information provided to ***ViewPort*** is in greater detail than the information recorded within the ASCII logging files. Also, ***ViewPort*** when an entry within an error window has been selected, will provide an explanation of the error received, both in terms of the normal error code and if applicable, the parameter error. Plate 2.1 shows ***ViewPort*** with two error windows and Plates 2.2 and 2.3 show the error information in detail.

Plate 2.1
ViewPort window



The detailed error information contained within the **Error Info** window shows a variety of information about the application including the process and thread ID's along with the date and time the error was recorded. In conjunction with this, when you have compiled your application to include source filename and line numbers with the API calls, the source file from which the API call was made will be displayed along with the line number. This helps you in correcting any non-expected errors from the API.

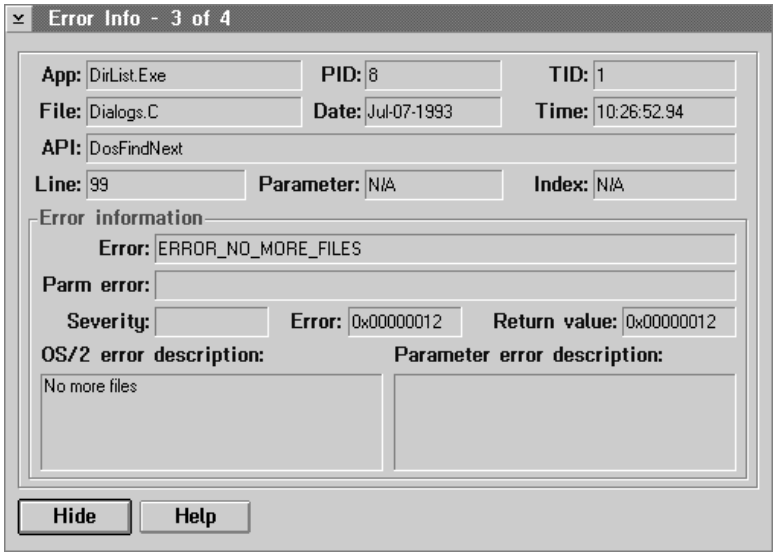
In the example, shown in Plate 2.2, the error returned by the **DosFindNext** call is expected. The ERROR_NO_MORE_FILES is used by the API to indicate that there are no further directory entries matching the original file search specification that was provided to the **DosFindFirst** API.

Since this is an error returned by the actual API itself, you will notice that the areas of the window for **Parameter**, **Index** , **Parm error** and **Parameter error description** are blank. These areas will only contain information when a parameter error has been detected by the validation

functions. An example of an error detected by the validation functions is shown in Plate 2.3.

Within the detail error information area, you will notice that the symbol used by OS/2 to denote the error is given in the **Error** area. The actual numeric error value is displayed in the second **Error** area and the return value, which in this case because of the API is the error value, is shown in the area labeled **Return value**. When an error description that provides further details on the error is available, it will be displayed in the area below the **OS/2 error description** label.

Plate 2.2
Error Info window

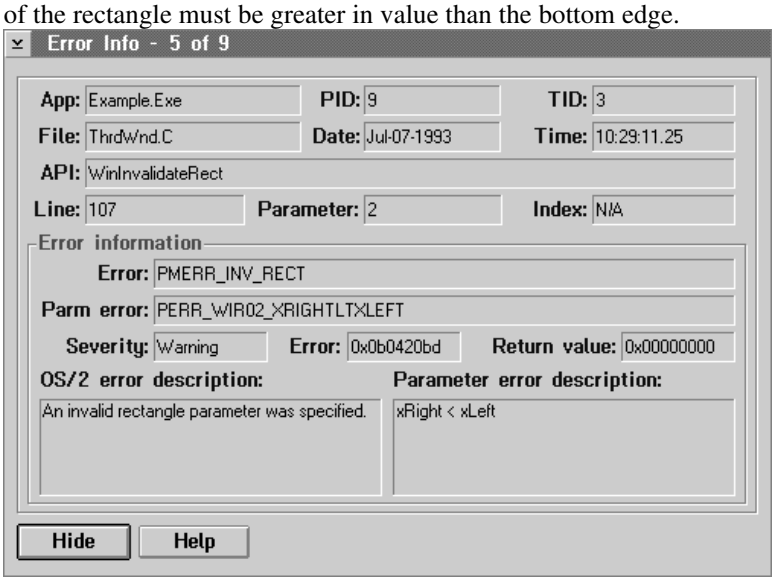


When a parameter error is detected by the validation routines, additional information is given to ***ViewPort***. This information deals with such things as the index of the parameter that is in error as well as the array index of the item in error if the parameter was an array.

In the example shown in Plate 2.3, you notice that the thread ID, **TID** is 3. The example application, which is in the Threads directory of the samples, invokes a thread which utilizes a loop with a time delay to update the main client window of the application to show the number of threads running. Through each iteration of the loop within the thread a display rectangle (**RECTL**) is completed where on odd iterations the rectangle is perfectly valid whereas on even iterations the *xLeft* and *xRight* values are exchanged such that validation routines pick up the discrepancy.

When the **WinInvalidateRect** function is called, the validation routines check the parameters to ensure that they are valid for the function. This includes making sure that the window handle is valid, the **RECTL** structure is a valid pointer and that the values of the structure are within the limits defined for the structure. It also checks to make sure that the values are valid in terms of a value that should be greater than the other. The example shows the case where the right edge of the rectangle having a value less than the left edge. By definition the right edge of the rectangle must always be greater than the left edge just like the top edge of the rectangle must be greater in value than the bottom edge.

Plate 2.3
Error information window showing a parameter error



In addition to the standard error information as described in the example above, the **Parm error** labeled area contains the symbol defined for the error which is numerically the high byte of the **Error** value. Also, the **Parameter error description** contains the explanation of the error.

For a complete description of how you use ***ViewPort***, Section 4, ***Using ViewPort*** (see page 63) should be consulted.

Preparing your source code for validation support

There are three ways of using ***Validator*** with your applications. The first method does not require any re-compilation of your source code. It only requires that you link your application with the VALIDATR.LIB library.

The second method requires that you make a minor change to your source code, usually in the module where your **main( )** function is located. Here you add the **ValInitialize** (see page 55) function which

allows you to utilize the advance features provided within the validation routines for logging the error information to logging files or by passing the information to ***ViewPort***. This module would then have to be recompiled such that the symbol INCL_VAL is defined before the OS2.H file is included by the module. The following shows the general structure of the module:

Figure 2.18 Validation API usage

```
include <os2.h>

INT main(INT argc, CHAR *argv[ ])

{
HVAL  hval;                        /* Validator Handle                        */

hval = ValInitialize("Example.Exe", "Example.Log", VL_ERRORLOG);
•
•
•
DosExit(EXIT_PROCESS, 0UL);
return(0);
}
```

When you compile the module, you would include the /DINCL_VAL switch with the other compiler options such that when the module is compiled, the necessary header files for the validation routines will be included through the OS2.H header.

This second level also requires that you link your final application with the VALIDATR.LIB library.

Depending on the options you used with the **ValInitialize** (see page 55) function, you may or may not require ***ViewPort*** to be running when you run your application after it has been linked.

The last method is similar to above, except that you need to recompile all of your source modules such that they define the INCL_VALAPI before the inclusion of the OS2.H header and you make changes to the main( ) like that shown in Figure 2.18. Again, you would have to link your application with the VALIDATR.LIB library.

To show you exactly what is required for each of the methods, the following simple application will be used to illustrate exactly what is required. It is assumed that the compiler to be used is the IBM C Set++ compiler. The linker is that provided by the OS/2 Toolkit. The starting source code is:

Figure 2.19 Simple example

```
#define INCL_WIN                   /* Include OS/2 PM Windows Interface       */

#include <os2.h>

int main( )

{
HAB    hAB;                        /* Anchor Block Handle                     */
HMQ    hmqExample;                 /* Message Queue Handle                    */

                       /* Initialize the program for PM and create the        */
```

```
                        /* message queue                                        */

hmqExample = WinCreateMsgQueue(hAB = WinInitialize(0UL), 0L);

WinMessageBox(HWND_DESKTOP, HWND_DESKTOP, "Hello world!!",
              "Validator Example", 0UL, MB_OK | MB_ICONEXCLAMATION | MB_MOVEABLE);

WinDestroyMsgQueue(hmqExample);

                        /* Notify PM that main program thread not needed        */
                        /* any longer                                           */
WinTerminate(hAB);
return(0);
}
```

The definition file will be the same for the normal and validation enabled steps.

Figure 2.20 Example.Def

```
NAME                    Example WINDOWAPI

DESCRIPTION             'Validator Example'

CODE                    MOVEABLE
DATA                    MULTIPLE MOVEABLE

STACKSIZE               8192

PROTMODE

EXETYPE                 OS2
```

The compile and link instructions that would normally be used are:

Figure 2.21 Compile and link instructions

```
icc -G3 -Ox -C -FoExample Example.C
link386 Example,Example, ,Example.Def;
```

For the first level of validation support, all you would need to do is change the compile and link step to:

Figure 2.22 Revised compile and link instructions

```
icc -G3 -Ox -C -FoExample Example.C
link386 Example,Example,Validatr.Lib,Example.Def;
```

This will permit the validation routines to check each OS/2 API and return both normal API errors and parameter errors. This is useful when debugging.

The second level requires that you compile the application after adding the validation API's. The resulting source code would be:

Figure 2.23 Revised example

```
#define INCL_WIN                    /* Include OS/2 PM Windows Interface       */

#include <os2.h>

int main( )

{
HAB  hAB;                           /* Anchor Block Handle                     */
HMQ  hmqExample;                    /* Message Queue Handle                    */
HVAL hval;                          /* Validation Handle                       */

hval = ValInitialize("Validation Example", "Example.Log",
                     VL_ERRORLOG | VL_VIEWPORT);

                        /* Initialize the program for PM and create the        */
                        /* message queue                                       */

hmqExample = WinCreateMsgQueue(hAB = WinInitialize(0UL), 0L);

WinMessageBox(HWND_DESKTOP, HWND_DESKTOP, "Hello world!!",
              "Validator Example", 0UL, MB_OK | MB_ICONEXCLAMATION | MB_MOVEABLE);
```

```
WinDestroyMsgQueue(hmqExample);

                    /* Notify PM that main program thread not needed       */
                    /* any longer                                          */
WinTerminate(hAB);
return(0);
}
```

The compile and link instructions that would change to:

Figure 2.24 Revised compile and link instructions

```
icc -G3 -Ox -C -DINCL_VAL -FoExample Example.C
link386 Example,Example,Validatr.Lib,Example.Def;
```

The support requested through **ValInitialize** was to have ASCII logging to the Example.Log file and the error information also sent to ***ViewPort***. Although, the changes allowed for error logging and ***ViewPort***, the inclusion symbol only requested base support through the validation routines. INCL_VAL does not enable the inclusion of the source filename and line number with each OS/2 API. The third example changes the compile and link instructions to be:

Figure 2.25 Revised compile and link instructions

```
icc -G3 -Ox -C -DINCL_VALAPI -FoExample Example.C
link386 Example,Example,Validatr.Lib,Example.Def;
```

This last change would enable the inclusion of the filename and line numbers such that when an error occurs, the filename of the source module and the line number are included with the logging information and are also provided to ***ViewPort***.

The files provided with ***Validator*** beyond the sample files are for the validation DLL's and the headers. The following DLL's are provided:

Table 2.1 Control DLL's

DLL	Purpose
VALIDATR.DLL	Validation controller.
API.DLL	API List.

Table 2.2 Validation DLL's

DLL	Purpose
VALDDF.DLL	Ddf* calls.
VALDEV.DLL	Dev* calls.
VALDOS.DLL	Dos* calls.
VALDOSP.DLL	Dos* Version 2.1 only calls.
VALDRG.DLL	Drg* calls.
VALGPI.DLL	Gpi* calls.
VALMCI.DLL	MMPM/2 calls where REXX is required.

DLL	Purpose
VALMMPM.DLL	MMPM/2 calls.
VALPEN.DLL	Pen for OS/2 calls.
VALPIC.DLL	Pic* calls.
VALPRF.DLL	Prf* calls.
ValPrt.DLL	Prt* calls.
VALSPL.DLL	Spl* calls.
VALWIN.DLL	Win* calls.
VALWINX.DLL	Win* Version 2.1 only calls.

Table 2.3
Validation DLL's which include filename and line numbers

DLL	Purpose
VALDDFA.DLL	Ddf* calls.
VALDEVA.DLL	Dev* calls.
VALDOSA.DLL	Dos* calls.
VALDOSPA.DLL	Dos* Version 2.1 only calls
VALDRGA.DLL	Drg* calls.
VALGPIA.DLL	Gpi* calls.
VALMCIA.DLL	MMPM/2 calls where REXX is required.
VALMMPMA.DLL	MMPM/2 calls.
VALPENA.DLL	Pen for OS/2 calls.
VALPICA.DLL	Pic* calls.
VALPRFA.DLL	Prf* calls.
VALPRTA.DLL	Prt* calls.
VALSPLA.DLL	Spl* calls.
VALWINA.DLL	Win* calls.
VALWINXA.DLL	Win* Version 2.1 only calls.

The following header files are provided:

Table 2.4
Validation headers

Header	Purpose
OS2.H	Replacement for toolkit OS2.H allowing for inclusion of validation headers.

Header	Purpose
OS2ME.H	Replacement for MMPM/2 toolkit OS2ME.H allowing for inclusion of validation headers.
PENPM.H	Replacement for Pen for OS/2 toolkit PENPM.H allowing for inclusion of validation headers.
VALAPI.H	Validation API and OS/2 API indices.
VALERRS.H	Validation parameter errors constants header.
VALFLINE.H	Validation definition including source filename and line numbers within OS/2 API prototypes.
VALREDEF.H	Validation OS/2 API macro redefinition header for inclusion of source filename and line within OS/2 API call.

API's

Introduction

Validator provides a set of API's that allow you to select the level of validation support and monitoring. You can utilize these features during the development process as well as when you are beta testing with third parties.

Six API calls are provided that allow you to setup the validation support level along with the ability to dictate specific errors for specific API's that should be filtered. The first API used is **ValInitialize** (see page 55) which allows you to setup the level of validation support beyond the additional parameter error information provided with the normal error codes. The filtering API, **ValFilterErr** (see page 30), allows you to specify which error codes to ignore for a given API. Logging is controlled through **ValLogging** (see page 57) and **ValQueryLogging** (see page 59). Monitoring of the message parameters within **WinSendDlgItemMsg** and **WinSendMsg** API's is defined through the **ValRegisterClassMsgMonitor** (see page 59) and **ValQueryClassMsgMonitor** (see page 58).

Generally you would use API's as follows:

Figure 3.1 API usage

```
INT main(INT argc, CHAR *argv[ ])

{
QMSG  qmsg;                        /* PM Message Queue Holder               */
HVAL hval;                         /* Validator Handle                      */

hval = ValInitialize("Example.Exe", NULL, VL_VIEWPORT);

                        /* Initialize the program for PM and create the     */
                        /* message queue                                    */

hmqExample = WinCreateMsgQueue(hAB = WinInitialize(0UL), 0L);
```

```
                        /* Register the window class                          */

if ( !WinRegisterClass(hAB, pszExampleClassName, (PFNWP)ExampleWndProc,
                        CS_SYNCPAINT | CS_SIZEREDRAW, 0UL) )
   return(0);
                        /* Create the main program window                     */

if ( !(hwndExampleFrame = CreateStdWindow(HWND_DESKTOP, WS_VISIBLE,
                                          FCF_NOBYTEALIGN | FCF_SHELLPOSITION |
                                          FCF_TITLEBAR | FCF__ICON |
                                          FCF_SYSMENU | FCF_MENU | FCF_MINMAX |
                                          FCF_ACCELTABLE | FCF_SIZEBORDER,
                                          pszExampleClassName,
                                          "Example Window", 0L,
                                          (HMODULE)NULL, ID_WINDOW,
                                          &hwndExample,
                                          10L, 23L, 337L, 196L)) )
   return(0);
InitProg( );
                        /* Get and dispatch the message to program            */
                        /* windows                                            */

while ( WinGetMsg(hAB, &qmsg, (HWND)NULL, 0UL, 0UL) )
   WinDispatchMsg(hAB, &qmsg);

                        /* Have received a WM_QUIT, start the program         */
                        /* shutdown by destroying the program windows         */
                        /* and destroying the message queue                   */

WinDestroyWindow(hwndExampleFrame);
WinDestroyMsgQueue(hmqExample);

                        /* Notify PM that main program thread not needed      */
                        /* any longer                                         */
WinTerminate(hAB);
                        /* Exit back to OS/2 cleanly                          */
DosExit(EXIT_PROCESS, 0UL);
return(0);
}
```

Upon entry into the **main**( ) function before you issue any calls, you would use the **ValInitialize** function to register the application with the validation DLL's. To allow the code to be easily adapted, you may want to utilize the fact that the symbol **INCL_VALAPI** or **INCL_VAL** needs to be defined. To allow the most flexibility, you would define it through the command line to the compiler you are using with the /D or -D switch. By bracketing the initialization and termination code with the #ifdef INCL_VALAPI and #endif statements, you can easily allow the source code to be used both for the final release versions and versions that are to be monitored.

In the example shown in Figure 3.2, the definition of the validation handle *hval* is placed within the #ifdef statement such that this is only defined when validation is to be provided for the application. Also within the #ifdef statement is the actual call to the **ValInitialize** initialization function. In this case, only ***ViewPort*** support is being requested.

If you were allowing for a beta release where the validation DLL's are provided as part of the beta and you want the usage of the applications to be monitored through logging files, you could do the following:

Figure 3.2 Logging file usage

```
INT main(INT argc, CHAR *argv[ ])
{
QMSG  qmsg;                        /* PM Message Queue Holder                */
#ifdef INCL_VALAPI
HVAL hval;                         /* Validator Handle                       */
hval = ValInitialize("Example.Exe", Example.Log, VL_VIEWPORTLOG);
#endif
```

Each time the application was run, the error information would be appended to the logging file. In this example, the logging file is in ***ViewPort*** format which means that the user would have to provide you with the file whereby you would use ***ViewPort*** to retrieve the file from disk for viewing. Alternately, if you use the VL_ERRORLOG in place of the VL_VIEWPORTLOG, then the file produced would be in ASCII.

As you can see, the usage of the actual API's is not that difficult. The following defines fully the usage of each of the validation API's available.

Linking considerations

When linking, you need to make sure that if you are using the /NOD switch with the LINK386 linker provided with the IBM OS/2 Toolkit, that you also in the OS2386.LIB ***after*** the Validatr.Lib otherwise you will receive unresolved externals. The /NOD switch prevents the linker from using the default libraries specified within the object modules produced by the C Set/2 or C Set++ compilers. Figure 3.3 shows an example of this:

Figure 3.2 Logging file usage

```
icc –G3 –Ox –C –DINCL_VALAPI –FoExample Example.C
link386 Example,Example,Validatr.Lib+OS2386.Lib,
Example.Def;
```

ValFilterErr

ULONG ValFilterErr*(hval, ulAPIFamily, ulAPI, ulError, ulReturn)*

```
HVAL hval;          /* Validation Handle   */
ULONG ulAPIFamily;  /* API Family          */
ULONG ulAPI;        /* API Designation     */
ULONG ulError;      /* OS/2 Error Value    */
ULONG ulReturn;     /* API Return Value    */
```

This function is used to add a specific API, error and return value to the filter list which is used to prevent these errors from being logged to a logging file or sent to ***ViewPort*** which could overload the system message queue.

Parameters

⇓ **HVAL** *hval*
Validation handle.

⇓ **ULONG** *ulAPIFamily*
API family designation. It can be one of the following values:

Symbol	Index	API
APIFAMILY_DDF	0x00000000	Ddf* calls
APIFAMILY_DEV	0x00010000	Dev* calls
APIFAMILY_DOS	0x00020000	Dos* calls
APIFAMILY_DRG	0x00030000	Drg* calls
APIFAMILY_GPI	0x00040000	Gpi* calls
APIFAMILY_PIC	0x00060000	Pic* calls
APIFAMILY_PRF	0x00070000	Prf* calls
APIFAMILY_PRT	0x00080000	Prt* calls
APIFAMILY_SPL	0x00090000	Spl* calls
APIFAMILY_WIN	0x000a0000	Win* calls
APIFAMILY_MMPM	0x000b0000	MMPM/2 calls
APIFAMILY_PEN	0x000c0000	Pen for OS/2 calls

⇓ **ULONG** *ulAPI*
API designation. Each designation is contained within an API family with each API having a designated value. The following are the valid values for the API designations:

Ddf* Calls

Symbol	Index	API
API_DDFBEGINLIST	0x0	DdfBeginList
API_DDFBITMAP	0x1	DdfBitmap
API_DDFENDLIST	0x2	DdfEndList
API_DDFHYPERTEXT	0x3	DdfHyperText
API_DDFINFORM	0x4	DdfInform

Symbol	Index	API
API_DDFINITIALIZE	0x5	DdfInitialize
API_DDFLISTITEM	0x6	DdfListItem
API_DDFMETAFILE	0x7	DdfMetafile
API_DDFPARA	0x8	DdfPara
API_DDFSETCOLOR	0x9	DdfSetColor
API_DDFSETFONT	0xa	DdfSetFont
API_DDFSETFONTSTYLE	0xb	DdfSetFontStyle
API_DDFSETFORMAT	0xc	DdfSetFormat
API_DDFSETTEXTALIGN	0xd	DdfSetTextAlign
API_DDFTEXT	0xe	DdfText

Dev* Calls

Symbol	Index	API
API_DEVCLOSEDC	0x0	DevCloseDC
API_DEVESCAPE	0x1	DevEscape
API_DEVOPENDC	0x2	DevOpenDC
API_DEVPOSTDEVICEMODES	0x3	DevPostDeviceModes
API_DEVQUERYCAPS	0x4	DevQueryCaps
API_DEVQUERYDEVICENAMES	0x5	DevQueryDeviceNames
API_DEVQUERYHARDCOPYCAPS	0x6	DevQuery-HardcopyCaps

Dos* Calls

Symbol	Index	API
API_DOSACKNOWLEDGESIGNALEXCEPTION	0x0	DosAcknowledge-SignalException
API_DOSADDMUXWAITSEM	0x1	DosAddMuxWaitSem
API_DOSALLOCMEM	0x2	DosAllocMem
API_DOSALLOCSHAREDMEM	0x3	DosAllocSharedMem
API_DOSASYNCTIMER	0x4	DosAsyncTimer
API_DOSBEEP	0x5	DosBeep
API_DOSCALLNPIPE	0x6	DosCallNPipe
API_DOSCANCELLOCKREQUEST	0x7	DosCancelLockRequest
API_DOSCLOSE	0x8	DosClose
API_DOSCLOSEEVENTSEM	0x9	DosCloseEventSem
API_DOSCLOSEMUTEXSEM	0xa	DosCloseMutexSem
API_DOSCLOSEMUXWAITSEM	0xb	DosCloseMuxWaitSem
API_DOSCLOSEQUEUE	0xc	DosCloseQueue
API_DOSCLOSEVDD	0xd	DosCloseVDD
API_DOSCONNECTNPIPE	0xe	DosConnectNPipe
API_DOSCOPY	0xf	DosCopy
API_DOSCREATEDIR	0x10	DosCreateDir
API_DOSCREATEEVENTSEM	0x11	DosCreateEventSem
API_DOSCREATEMUTEXSEM	0x12	DosCreateMutexSem
API_DOSCREATEMUXWAITSEM	0x13	DosCreateMuxWaitSem
API_DOSCREATENPIPE	0x14	DosCreateNPipe
API_DOSCREATEPIPE	0x15	DosCreatePipe
API_DOSCREATEQUEUE	0x16	DosCreateQueue

Symbol	Index	API
API_DOSCREATETHREAD	0x17	DosCreateThread
API_DOSDEBUG	0x18	DosDebug
API_DOSDELETE	0x19	DosDelete
API_DOSDELETEDIR	0x1a	DosDeleteDir
API_DOSDELETEMUXWAITSEM	0x1b	DosDeleteMuxWaitSem
API_DOSDEVCONFIG	0x1c	DosDevConfig
API_DOSDEVIOCTL	0x1d	DosDevIOCtl
API_DOSDISCONNECTNPIPE	0x1e	DosDisConnectNPipe
API_DOSDUPHANDLE	0x1f	DosDupHandle
API_DOSEDITNAME	0x20	DosEditName
API_DOSENTERCRITSEC	0x21	DosEnterCritSec
API_DOSENTERMUSTCOMPLETE	0x22	DosEnterMustComplete
API_DOSENUMATTRIBUTE	0x23	DosEnumAttribute
API_DOSERRCLASS	0x24	DosErrClass
API_DOSERROR	0x25	DosError
API_DOSEXECPGM	0x26	DosExecPgm
API_DOSEXIT	0x27	DosExit
API_DOSEXITCRITSEC	0x28	DosExitCritSec
API_DOSEXITLIST	0x29	DosExitList
API_DOSEXITMUSTCOMPLETE	0x2a	DosExitMustComplete
API_DOSFSATTACH	0x2b	DosFSAttach
API_DOSFSCTL	0x2c	DosFSCtl
API_DOSFINDCLOSE	0x2d	DosFindClose
API_DOSFINDFIRST	0x2e	DosFindFirst
API_DOSFINDNEXT	0x2f	DosFindNext
API_DOSFORCEDELETE	0x30	DosForceDelete
API_DOSFREEMEM	0x31	DosFreeMem
API_DOSFREEMODULE	0x32	DosFreeModule
API_DOSFREERESOURCE	0x33	DosFreeResource
API_DOSGETDATETIME	0x34	DosGetDateTime
API_DOSGETINFOBLOCKS	0x35	DosGetInfoBlocks
API_DOSGETMESSAGE	0x36	DosGetMessage
API_DOSGETNAMEDSHAREDMEM	0x37	DosGetNamed-SharedMem
API_DOSGETRESOURCE	0x38	DosGetResource
API_DOSGETSHAREDMEM	0x39	DosGetSharedMem
API_DOSGIVESHAREDMEM	0x3a	DosGiveSharedMem
API_DOSINSERTMESSAGE	0x3b	DosInsertMessage
API_DOSKILLPROCESS	0x3c	DosKillProcess
API_DOSKILLTHREAD	0x3d	DosKillThread
API_DOSLOADMODULE	0x3e	DosLoadModule
API_DOSMAPCASE	0x3f	DosMapCase
API_DOSMOVE	0x40	DosMove
API_DOSOPEN	0x41	DosOpen
API_DOSOPENEVENTSEM	0x42	DosOpenEventSem
API_DOSOPENMUTEXSEM	0x43	DosOpenMutexSem
API_DOSOPENMUXWAITSEM	0x44	DosOpenMuxWaitSem
API_DOSOPENQUEUE	0x45	DosOpenQueue
API_DOSOPENVDD	0x46	DosOpenVDD
API_DOSPEEKNPIPE	0x47	DosPeekNPipe
API_DOSPEEKQUEUE	0x48	DosPeekQueue

Symbol	Index	API
API_DOSPHYSICALDISK	0x49	DosPhysicalDisk
API_DOSPOSTEVENTSEM	0x4a	DosPostEventSem
API_DOSPURGEQUEUE	0x4b	DosPurgeQueue
API_DOSPUTMESSAGE	0x4c	DosPutMessage
API_DOSQUERYAPPTYPE	0x4d	DosQueryAppType
API_DOSQUERYCOLLATE	0x4e	DosQueryCollate
API_DOSQUERYCP	0x4f	DosQueryCp
API_DOSQUERYCTRYINFO	0x50	DosQueryCtryInfo
API_DOSQUERYCURRENTDIR	0x51	DosQueryCurrentDir
API_DOSQUERYCURRENTDISK	0x52	DosQueryCurrentDisk
API_DOSQUERYDBCSENV	0x53	DosQueryDBCSEnv
API_DOSQUERYEVENTSEM	0x54	DosQueryEventSem
API_DOSQUERYFHSTATE	0x55	DosQueryFHState
API_DOSQUERYFSATTACH	0x56	DosQueryFSAttach
API_DOSQUERYFSINFO	0x57	DosQueryFSInfo
API_DOSQUERYFILEINFO	0x58	DosQueryFileInfo
API_DOSQUERYHTYPE	0x59	DosQueryHType
API_DOSQUERYMEM	0x5a	DosQueryMem
API_DOSQUERYMESSAGECP	0x5b	DosQueryMessageCP
API_DOSQUERYMODULEHANDLE	0x5c	DosQueryModule-Handle
API_DOSQUERYMODULENAME	0x5d	DosQueryModuleName
API_DOSQUERYMUTEXSEM	0x5e	DosQueryMutexSem
API_DOSQUERYMUXWAITSEM	0x5f	DosQueryMuxWaitSem
API_DOSQUERYNPHSTATE	0x60	DosQueryNPHState
API_DOSQUERYNPIPEINFO	0x61	DosQueryNPipeInfo
API_DOSQUERYNPIPESEMSTATE	0x62	DosQueryNPipe-SemState
API_DOSQUERYPATHINFO	0x63	DosQueryPathInfo
API_DOSQUERYPROCADDR	0x64	DosQueryProcAddr
API_DOSQUERYPROCTYPE	0x65	DosQueryProcType
API_DOSQUERYQUEUE	0x66	DosQueryQueue
API_DOSQUERYRASINFO	0x67	DosQueryRASInfo
API_DOSQUERYRESOURCESIZE	0x68	DosQueryResourceSize
API_DOSQUERYSYSINFO	0x69	DosQuerySysInfo
API_DOSQUERYVERIFY	0x6a	DosQueryVerify
API_DOSRAISEEXCEPTION	0x6b	DosRaiseException
API_DOSRAWREADNPIPE	0x6c	DosRawReadNPipe
API_DOSRAWWRITENPIPE	0x6d	DosRawWriteNPipe
API_DOSREAD	0x6e	DosRead
API_DOSREADQUEUE	0x6f	DosReadQueue
API_DOSREGISTERPERFCTRS	0x70	DosRegisterPerfCtrs
API_DOSRELEASEMUTEXSEM	0x71	DosReleaseMutexSem
API_DOSREQUESTMUTEXSEM	0x72	DosRequestMutexSem
API_DOSREQUESTVDD	0x73	DosRequestVDD
API_DOSRESETBUFFER	0x74	DosResetBuffer
API_DOSRESETEVENTSEM	0x75	DosResetEventSem
API_DOSRESUMETHREAD	0x76	DosResumeThread
API_DOSSCANENV	0x77	DosScanEnv
API_DOSSEARCHPATH	0x78	DosSearchPath
API_DOSSELECTSESSION	0x79	DosSelectSession

Symbol	Index	API
API_DOSSENDSIGNALEXCEPTION	0x7a	DosSendSignal-Exception
API_DOSSETCURRENTDIR	0x7b	DosSetCurrentDir
API_DOSSETDOSPROPERTY	0x7c	DosSetDOSProperty
API_DOSSETDATETIME	0x7d	DosSetDateTime
API_DOSSETDEFAULTDISK	0x7e	DosSetDefaultDisk
API_DOSSETEXCEPTIONHANDLER	0x7f	DosSetException-Handler
API_DOSSETFHSTATE	0x80	DosSetFHState
API_DOSSETFSINFO	0x81	DosSetFSInfo
API_DOSSETFILEINFO	0x82	DosSetFileInfo
API_DOSSETFILELOCKS	0x83	DosSetFileLocks
API_DOSSETFILEPTR	0x84	DosSetFilePtr
API_DOSSETFILESIZE	0x85	DosSetFileSize
API_DOSSETMAXFH	0x86	DosSetMaxFH
API_DOSSETMEM	0x87	DosSetMem
API_DOSSETNPHSTATE	0x88	DosSetNPHState
API_DOSSETNPIPESEM	0x89	DosSetNPipeSem
API_DOSSETPATHINFO	0x8a	DosSetPathInfo
API_DOSSETPRIORITY	0x8b	DosSetPriority
API_DOSSETPROCESSCP	0x8c	DosSetProcessCp
API_DOSSETRELMAXFH	0x8d	DosSetRelMaxFH
API_DOSSETSESSION	0x8e	DosSetSession
API_DOSSETSIGNALEXCEPTIONFOCUS	0x8f	DosSetSignal-ExceptionFocus
API_DOSSETVERIFY	0x90	DosSetVerify
API_DOSSHUTDOWN	0x91	DosShutdown
API_DOSSLEEP	0x92	DosSleep
API_DOSSTARTSESSION	0x93	DosStartSession
API_DOSSTARTTIMER	0x94	DosStartTimer
API_DOSSTOPSESSION	0x95	DosStopSession
API_DOSSTOPTIMER	0x96	DosStopTimer
API_DOSSUBALLOCMEM	0x97	DosSubAllocMem
API_DOSSUBFREEMEM	0x98	DosSubFreeMem
API_DOSSUBSETMEM	0x99	DosSubSetMem
API_DOSSUBUNSETMEM	0x9a	DosSubUnsetMem
API_DOSSUSPENDTHREAD	0x9b	DosSuspendThread
API_DOSTMRQUERYFREQ	0x9c	DosTmrQueryFreq
API_DOSTMRQUERYTIME	0x9d	DosTmrQueryTime
API_DOSTRANSACTNPIPE	0x9e	DosTransactNPipe
API_DOSUNSETEXCEPTIONHANDLER	0x9f	DosUnset-ExceptionHandler
API_DOSUNWINDEXCEPTION	0xa0	DosUnwindException
API_DOSWAITCHILD	0xa1	DosWaitChild
API_DOSWAITEVENTSEM	0xa2	DosWaitEventSem
API_DOSWAITMUXWAITSEM	0xa3	DosWaitMuxWaitSem
API_DOSWAITNPIPE	0xa4	DosWaitNPipe
API_DOSWAITTHREAD	0xa5	DosWaitThread
API_DOSWRITE	0xa6	DosWrite
API_DOSWRITEQUEUE	0xa7	DosWriteQueue

OS/2 2.1 Dos Calls*

Symbol	Index	API
API_DOSPROTECTCLOSE	0xa8	DosProtectClose
API_DOSPROTECTENUMATTRIBUTE	0xa9	DosProtectEnum-Attribute
API_DOSPROTECTOPEN	0xaa	DosProtectOpen
API_DOSPROTECTQUERYFHSTATE	0xab	DosProtectQuery-FHState
API_DOSPROTECTQUERYFILEINFO	0xac	DosProtectQuery-FileInfo
API_DOSPROTECTREAD	0xad	DosProtectRead
API_DOSPROTECTSETFHSTATE	0xae	DosProtectSetFHState
API_DOSPROTECTSETFILEINFO	0xaf	DosProtectSetFileInfo
API_DOSPROTECTSETFILELOCKS	0xb0	DosProtectSetFileLocks
API_DOSPROTECTSETFILEPTR	0xb1	DosProtectSetFilePtr
API_DOSPROTECTSETFILESIZE	0xb2	DosProtectSetFileSize
API_DOSPROTECTWRITE	0xb3	DosProtectWrite

Drg Calls*

Symbol	Index	API
API_DRGACCEPTDROPPEDFILES	0x0	DrgAcceptDroppedFiles
API_DRGACCESSDRAGINFO	0x1	DrgAccessDraginfo
API_DRGADDSTRHANDLE	0x2	DrgAddStrHandle
API_DRGALLOCDRAGINFO	0x3	DrgAllocDraginfo
API_DRGALLOCDRAGTRANSFER	0x4	DrgAllocDragtransfer
API_DRGDELETEDRAGINFOSTRHANDLES	0x5	DrgDeleteDraginfo-StrHandles
API_DRGDELETESTRHANDLE	0x6	DrgDeleteStrHandle
API_DRGDRAG	0x7	DrgDrag
API_DRGDRAGFILES	0x8	DrgDragFiles
API_DRGFREEDRAGINFO	0x9	DrgFreeDraginfo
API_DRGFREEDRAGTRANSFER	0xa	DrgFreeDragtransfer
API_DRGGETPS	0xb	DrgGetPS
API_DRGPOSTTRANSFERMSG	0xc	DrgPostTransferMsg
API_DRGPUSHDRAGINFO	0xd	DrgPushDraginfo
API_DRGQUERYDRAGITEM	0xe	DrgQueryDragitem
API_DRGQUERYDRAGITEMCOUNT	0xf	DrgQueryDragitem-Count
API_DRGQUERYDRAGITEMPTR	0x10	DrgQueryDragitemPtr
API_DRGQUERYNATIVERMF	0x11	DrgQueryNativeRMF
API_DRGQUERYNATIVERMFLEN	0x12	DrgQueryNative-RMFLen
API_DRGQUERYSTRNAME	0x13	DrgQueryStrName
API_DRGQUERYSTRNAMELEN	0x14	DrgQueryStrNameLen
API_DRGQUERYTRUETYPE	0x15	DrgQueryTrueType
API_DRGQUERYTRUETYPELEN	0x16	DrgQueryTrueTypeLen
API_DRGRELEASEPS	0x17	DrgReleasePS
API_DRGSENDTRANSFERMSG	0x18	DrgSendTransferMsg
API_DRGSETDRAGIMAGE	0x19	DrgSetDragImage

Symbol	Index	API
API_DRGSETDRAGPOINTER	0x1a	DrgSetDragPointer
API_DRGSETDRAGITEM	0x1b	DrgSetDragitem
API_DRGVERIFYNATIVERMF	0x1c	DrgVerifyNativeRMF
API_DRGVERIFYRMF	0x1d	DrgVerifyRMF
API_DRGVERIFYTRUETYPE	0x1e	DrgVerifyTrueType
API_DRGVERIFYTYPE	0x1f	DrgVerifyType
API_DRGVERIFYTYPESET	0x20	DrgVerifyTypeSet

Gpi* Calls

Symbol	Index	API
API_GPIANIMATEPALETTE	0x0	GpiAnimatePalette
API_GPIASSOCIATE	0x1	GpiAssociate
API_GPIBEGINAREA	0x2	GpiBeginArea
API_GPIBEGINELEMENT	0x3	GpiBeginElement
API_GPIBEGINPATH	0x4	GpiBeginPath
API_GPIBITBLT	0x5	GpiBitBlt
API_GPIBOX	0x6	GpiBox
API_GPICALLSEGMENTMATRIX	0x7	GpiCallSegmentMatrix
API_GPICHARSTRING	0x8	GpiCharString
API_GPICHARSTRINGAT	0x9	GpiCharStringAt
API_GPICHARSTRINGPOS	0xa	GpiCharStringPos
API_GPICHARSTRINGPOSAT	0xb	GpiCharStringPosAt
API_GPICLOSEFIGURE	0xc	GpiCloseFigure
API_GPICLOSESEGMENT	0xd	GpiCloseSegment
API_GPICOMBINEREGION	0xe	GpiCombineRegion
API_GPICOMMENT	0xf	GpiComment
API_GPICONVERT	0x10	GpiConvert
API_GPICONVERTWITHMATRIX	0x11	GpiConvertWithMatrix
API_GPICOPYMETAFILE	0x12	GpiCopyMetaFile
API_GPICORRELATECHAIN	0x13	GpiCorrelateChain
API_GPICORRELATEFROM	0x14	GpiCorrelateFrom
API_GPICORRELATESEGMENT	0x15	GpiCorrelateSegment
API_GPICREATEBITMAP	0x16	GpiCreateBitmap
API_GPICREATELOGCOLORTABLE	0x17	GpiCreateLog-ColorTable
API_GPICREATELOGFONT	0x18	GpiCreateLogFont
API_GPICREATEPS	0x19	GpiCreatePS
API_GPICREATEPALETTE	0x1a	GpiCreatePalette
API_GPICREATEREGION	0x1b	GpiCreateRegion
API_GPIDELETEBITMAP	0x1c	GpiDeleteBitmap
API_GPIDELETEELEMENT	0x1d	GpiDeleteElement
API_GPIDELETEELEMENTRANGE	0x1e	GpiDeleteElementRange
API_GPIDELETEELEMENTSBETWEENLABELS	0x1f	GpiDelete-ElementsBetweenLabels
API_GPIDELETEMETAFILE	0x20	GpiDeleteMetaFile
API_GPIDELETEPALETTE	0x21	GpiDeletePalette
API_GPIDELETESEGMENT	0x22	GpiDeleteSegment
API_GPIDELETESEGMENTS	0x23	GpiDeleteSegments
API_GPIDELETESETID	0x24	GpiDeleteSetId

Symbol	Index	API
API_GPIDESTROYPS	0x25	GpiDestroyPS
API_GPIDESTROYREGION	0x26	GpiDestroyRegion
API_GPIDRAWBITS	0x27	GpiDrawBits
API_GPIDRAWCHAIN	0x28	GpiDrawChain
API_GPIDRAWDYNAMICS	0x29	GpiDrawDynamics
API_GPIDRAWFROM	0x2a	GpiDrawFrom
API_GPIDRAWSEGMENT	0x2b	GpiDrawSegment
API_GPIELEMENT	0x2c	GpiElement
API_GPIENDAREA	0x2d	GpiEndArea
API_GPIENDELEMENT	0x2e	GpiEndElement
API_GPIENDPATH	0x2f	GpiEndPath
API_GPIEQUALREGION	0x30	GpiEqualRegion
API_GPIERASE	0x31	GpiErase
API_GPIERRORSEGMENTDATA	0x32	GpiErrorSegmentData
API_GPIEXCLUDECLIPRECTANGLE	0x33	GpiExcludeClip-Rectangle
API_GPIFILLPATH	0x34	GpiFillPath
API_GPIFLOODFILL	0x35	GpiFloodFill
API_GPIFRAMEREGION	0x36	GpiFrameRegion
API_GPIFULLARC	0x37	GpiFullArc
API_GPIGETDATA	0x38	GpiGetData
API_GPIIMAGE	0x39	GpiImage
API_GPIINTERSECTCLIPRECTANGLE	0x3a	GpiIntersectClip-Rectangle
API_GPILABEL	0x3b	GpiLabel
API_GPILINE	0x3c	GpiLine
API_GPILOADBITMAP	0x3d	GpiLoadBitmap
API_GPILOADFONTS	0x3e	GpiLoadFonts
API_GPILOADMETAFILE	0x3f	GpiLoadMetaFile
API_GPILOADPUBLICFONTS	0x40	GpiLoadPublicFonts
API_GPIMARKER	0x41	GpiMarker
API_GPIMODIFYPATH	0x42	GpiModifyPath
API_GPIMOVE	0x43	GpiMove
API_GPIOFFSETCLIPREGION	0x44	GpiOffsetClipRegion
API_GPIOFFSETELEMENTPOINTER	0x45	GpiOffsetElement-Pointer
API_GPIOFFSETREGION	0x46	GpiOffsetRegion
API_GPIOPENSEGMENT	0x47	GpiOpenSegment
API_GPIOUTLINEPATH	0x48	GpiOutlinePath
API_GPIPAINTREGION	0x49	GpiPaintRegion
API_GPIPARTIALARC	0x4a	GpiPartialArc
API_GPIPATHTOREGION	0x4b	GpiPathToRegion
API_GPIPLAYMETAFILE	0x4c	GpiPlayMetaFile
API_GPIPOINTARC	0x4d	GpiPointArc
API_GPIPOLYFILLET	0x4e	GpiPolyFillet
API_GPIPOLYFILLETSHARP	0x4f	GpiPolyFilletSharp
API_GPIPOLYLINE	0x50	GpiPolyLine
API_GPIPOLYLINEDISJOINT	0x51	GpiPolyLineDisjoint
API_GPIPOLYMARKER	0x52	GpiPolyMarker
API_GPIPOLYSPLINE	0x53	GpiPolySpline
API_GPIPOLYGONS	0x54	GpiPolygons

Symbol	Index	API
API_GPIPOP	0x55	GpiPop
API_GPIPTINREGION	0x56	GpiPtInRegion
API_GPIPTVISIBLE	0x57	GpiPtVisible
API_GPIPUTDATA	0x58	GpiPutData
API_GPIQUERYARCPARAMS	0x59	GpiQueryArcParams
API_GPIQUERYATTRMODE	0x5a	GpiQueryAttrMode
API_GPIQUERYATTRS	0x5b	GpiQueryAttrs
API_GPIQUERYBACKCOLOR	0x5c	GpiQueryBackColor
API_GPIQUERYBACKMIX	0x5d	GpiQueryBackMix
API_GPIQUERYBITMAPBITS	0x5e	GpiQueryBitmapBits
API_GPIQUERYBITMAPDIMENSION	0x5f	GpiQueryBitmap-Dimension
API_GPIQUERYBITMAPHANDLE	0x60	GpiQueryBitmapHandle
API_GPIQUERYBITMAPINFOHEADER	0x61	GpiQueryBitmap-InfoHeader
API_GPIQUERYBITMAPPARAMETERS	0x62	GpiQueryBitmap-Parameters
API_GPIQUERYBOUNDARYDATA	0x63	GpiQueryBoundaryData
API_GPIQUERYCHARANGLE	0x64	GpiQueryCharAngle
API_GPIQUERYCHARBOX	0x65	GpiQueryCharBox
API_GPIQUERYCHARBREAKEXTRA	0x66	GpiQueryChar-BreakExtra
API_GPIQUERYCHARDIRECTION	0x67	GpiQueryCharDirection
API_GPIQUERYCHAREXTRA	0x68	GpiQueryCharExtra
API_GPIQUERYCHARMODE	0x69	GpiQueryCharMode
API_GPIQUERYCHARSET	0x6a	GpiQueryCharSet
API_GPIQUERYCHARSHEAR	0x6b	GpiQueryCharShear
API_GPIQUERYCHARSTRINGPOS	0x6c	GpiQueryCharStringPos
API_GPIQUERYCHARSTRINGPOSAT	0x6d	GpiQueryChar-StringPosAt
API_GPIQUERYCLIPBOX	0x6e	GpiQueryClipBox
API_GPIQUERYCLIPREGION	0x6f	GpiQueryClipRegion
API_GPIQUERYCOLOR	0x70	GpiQueryColor
API_GPIQUERYCOLORDATA	0x71	GpiQueryColorData
API_GPIQUERYCOLORINDEX	0x72	GpiQueryColorIndex
API_GPIQUERYCP	0x73	GpiQueryCp
API_GPIQUERYCURRENTPOSITION	0x74	GpiQueryCurrent-Position
API_GPIQUERYDEFARCPARAMS	0x75	GpiQueryDefArcParams
API_GPIQUERYDEFATTRS	0x76	GpiQueryDefAttrs
API_GPIQUERYDEFCHARBOX	0x77	GpiQueryDefCharBox
API_GPIQUERYDEFTAG	0x78	GpiQueryDefTag
API_GPIQUERYDEFVIEWINGLIMITS	0x79	GpiQueryDef-ViewingLimits
API_GPIQUERYDEFAULTVIEWMATRIX	0x7a	GpiQueryDefault-ViewMatrix
API_GPIQUERYDEVICE	0x7b	GpiQueryDevice
API_GPIQUERYDEVICEBITMAPFORMATS	0x7c	GpiQueryDevice-BitmapFormats
API_GPIQUERYDRAWCONTROL	0x7d	GpiQueryDrawControl
API_GPIQUERYDRAWINGMODE	0x7e	GpiQueryDrawingMode

Symbol	Index	API
API_GPIQUERYEDITMODE	0x7f	GpiQueryEditMode
API_GPIQUERYELEMENT	0x80	GpiQueryElement
API_GPIQUERYELEMENTPOINTER	0x81	GpiQueryElement-Pointer
API_GPIQUERYELEMENTTYPE	0x82	GpiQueryElementType
API_GPIQUERYFACESTRING	0x83	GpiQueryFaceString
API_GPIQUERYFONTACTION	0x84	GpiQueryFontAction
API_GPIQUERYFONTFILEDESCRIPTIONS	0x85	GpiQueryFont-FileDescriptions
API_GPIQUERYFONTMETRICS	0x86	GpiQueryFontMetrics
API_GPIQUERYFONTS	0x87	GpiQueryFonts
API_GPIQUERYFULLFONTFILEDESCS	0x88	GpiQueryFull-FontFileDescs
API_GPIQUERYGRAPHICSFIELD	0x89	GpiQueryGraphicsField
API_GPIQUERYINITIALSEGMENTATTRS	0x8a	GpiQueryInitial-SegmentAttrs
API_GPIQUERYKERNINGPAIRS	0x8b	GpiQueryKerningPairs
API_GPIQUERYLINEEND	0x8c	GpiQueryLineEnd
API_GPIQUERYLINEJOIN	0x8d	GpiQueryLineJoin
API_GPIQUERYLINETYPE	0x8e	GpiQueryLineType
API_GPIQUERYLINEWIDTH	0x8f	GpiQueryLineWidth
API_GPIQUERYLINEWIDTHGEOM	0x90	GpiQueryLine-WidthGeom
API_GPIQUERYLOGCOLORTABLE	0x91	GpiQueryLog-ColorTable
API_GPIQUERYLOGICALFONT	0x92	GpiQueryLogicalFont
API_GPIQUERYMARKER	0x93	GpiQueryMarker
API_GPIQUERYMARKERBOX	0x94	GpiQueryMarkerBox
API_GPIQUERYMARKERSET	0x95	GpiQueryMarkerSet
API_GPIQUERYMETAFILEBITS	0x96	GpiQueryMetaFileBits
API_GPIQUERYMETAFILELENGTH	0x97	GpiQueryMeta-FileLength
API_GPIQUERYMIX	0x98	GpiQueryMix
API_GPIQUERYMODELTRANSFORMMATRIX	0x99	GpiQueryModel-TransformMatrix
API_GPIQUERYNEARESTCOLOR	0x9a	GpiQueryNearestColor
API_GPIQUERYNUMBERSETIDS	0x9b	GpiQueryNumberSetIds
API_GPIQUERYPS	0x9c	GpiQueryPS
API_GPIQUERYPAGEVIEWPORT	0x9d	GpiQueryPageViewport
API_GPIQUERYPALETTE	0x9e	GpiQueryPalette
API_GPIQUERYPALETTEINFO	0x9f	GpiQueryPaletteInfo
API_GPIQUERYPATTERN	0xa0	GpiQueryPattern
API_GPIQUERYPATTERNREFPOINT	0xa1	GpiQueryPattern-RefPoint
API_GPIQUERYPATTERNSET	0xa2	GpiQueryPatternSet
API_GPIQUERYPEL	0xa3	GpiQueryPel
API_GPIQUERYPICKAPERTUREPOSITION	0xa4	GpiQueryPick-AperturePosition
API_GPIQUERYPICKAPERTURESIZE	0xa5	GpiQueryPick-ApertureSize
API_GPIQUERYRGBCOLOR	0xa6	GpiQueryRGBColor

Symbol	Index	API
API_GPIQUERYREALCOLORS	0xa7	GpiQueryRealColors
API_GPIQUERYREGIONBOX	0xa8	GpiQueryRegionBox
API_GPIQUERYREGIONRECTS	0xa9	GpiQueryRegionRects
API_GPIQUERYSEGMENTATTRS	0xaa	GpiQuerySegmentAttrs
API_GPIQUERYSEGMENTNAMES	0xab	GpiQuery-SegmentNames
API_GPIQUERYSEGMENTPRIORITY	0xac	GpiQuery-SegmentPriority
API_GPIQUERYSEGMENTTRANSFORMMATRIX	0xad	GpiQuerySegment-TransformMatrix
API_GPIQUERYSETIDS	0xae	GpiQuerySetIds
API_GPIQUERYSTOPDRAW	0xaf	GpiQueryStopDraw
API_GPIQUERYTAG	0xb0	GpiQueryTag
API_GPIQUERYTEXTALIGNMENT	0xb1	GpiQueryText-Alignment
API_GPIQUERYTEXTBOX	0xb2	GpiQueryTextBox
API_GPIQUERYVIEWINGLIMITS	0xb3	GpiQueryViewingLimits
API_GPIQUERYVIEWINGTRANSFORMMATRIX	0xb4	GpiQueryViewing-TransformMatrix
API_GPIQUERYWIDTHTABLE	0xb5	GpiQueryWidthTable
API_GPIRECTINREGION	0xb6	GpiRectInRegion
API_GPIRECTVISIBLE	0xb7	GpiRectVisible
API_GPIREMOVEDYNAMICS	0xb8	GpiRemoveDynamics
API_GPIRESETBOUNDARYDATA	0xb9	GpiResetBoundaryData
API_GPIRESETPS	0xba	GpiResetPS
API_GPIRESTOREPS	0xbb	GpiRestorePS
API_GPIROTATE	0xbc	GpiRotate
API_GPISAVEMETAFILE	0xbd	GpiSaveMetaFile
API_GPISAVEPS	0xbe	GpiSavePS
API_GPISCALE	0xbf	GpiScale
API_GPISELECTPALETTE	0xc0	GpiSelectPalette
API_GPISETARCPARAMS	0xc1	GpiSetArcParams
API_GPISETATTRMODE	0xc2	GpiSetAttrMode
API_GPISETATTRS	0xc3	GpiSetAttrs
API_GPISETBACKCOLOR	0xc4	GpiSetBackColor
API_GPISETBACKMIX	0xc5	GpiSetBackMix
API_GPISETBITMAP	0xc6	GpiSetBitmap
API_GPISETBITMAPBITS	0xc7	GpiSetBitmapBits
API_GPISETBITMAPDIMENSION	0xc8	GpiSetBitmap-Dimension
API_GPISETBITMAPID	0xc9	GpiSetBitmapId
API_GPISETCHARANGLE	0xca	GpiSetCharAngle
API_GPISETCHARBOX	0xcb	GpiSetCharBox
API_GPISETCHARBREAKEXTRA	0xcc	GpiSetCharBreakExtra
API_GPISETCHARDIRECTION	0xcd	GpiSetCharDirection
API_GPISETCHAREXTRA	0xce	GpiSetCharExtra
API_GPISETCHARMODE	0xcf	GpiSetCharMode
API_GPISETCHARSET	0xd0	GpiSetCharSet
API_GPISETCHARSHEAR	0xd1	GpiSetCharShear
API_GPISETCLIPPATH	0xd2	GpiSetClipPath
API_GPISETCLIPREGION	0xd3	GpiSetClipRegion

Symbol	Index	API
API_GPISETCOLOR	0xd4	GpiSetColor
API_GPISETCP	0xd5	GpiSetCp
API_GPISETCURRENTPOSITION	0xd6	GpiSetCurrentPosition
API_GPISETDEFARCPARAMS	0xd7	GpiSetDefArcParams
API_GPISETDEFATTRS	0xd8	GpiSetDefAttrs
API_GPISETDEFTAG	0xd9	GpiSetDefTag
API_GPISETDEFVIEWINGLIMITS	0xda	GpiSetDef-ViewingLimits
API_GPISETDEFAULTVIEWMATRIX	0xdb	GpiSetDefault-ViewMatrix
API_GPISETDRAWCONTROL	0xdc	GpiSetDrawControl
API_GPISETDRAWINGMODE	0xdd	GpiSetDrawingMode
API_GPISETEDITMODE	0xde	GpiSetEditMode
API_GPISETELEMENTPOINTER	0xdf	GpiSetElementPointer
API_GPISETELEMENTPOINTERATLABEL	0xe0	GpiSetElement-PointerAtLabel
API_GPISETGRAPHICSFIELD	0xe1	GpiSetGraphicsField
API_GPISETINITIALSEGMENTATTRS	0xe2	GpiSetInitial-SegmentAttrs
API_GPISETLINEEND	0xe3	GpiSetLineEnd
API_GPISETLINEJOIN	0xe4	GpiSetLineJoin
API_GPISETLINETYPE	0xe5	GpiSetLineType
API_GPISETLINEWIDTH	0xe6	GpiSetLineWidth
API_GPISETLINEWIDTHGEOM	0xe7	GpiSetLineWidthGeom
API_GPISETMARKER	0xe8	GpiSetMarker
API_GPISETMARKERBOX	0xe9	GpiSetMarkerBox
API_GPISETMARKERSET	0xea	GpiSetMarkerSet
API_GPISETMETAFILEBITS	0xeb	GpiSetMetaFileBits
API_GPISETMIX	0xec	GpiSetMix
API_GPISETMODELTRANSFORMMATRIX	0xed	GpiSetModel-TransformMatrix
API_GPISETPS	0xee	GpiSetPS
API_GPISETPAGEVIEWPORT	0xef	GpiSetPageViewport
API_GPISETPALETTEENTRIES	0xf0	GpiSetPaletteEntries
API_GPISETPATTERN	0xf1	GpiSetPattern
API_GPISETPATTERNREFPOINT	0xf2	GpiSetPatternRefPoint
API_GPISETPATTERNSET	0xf3	GpiSetPatternSet
API_GPISETPEL	0xf4	GpiSetPel
API_GPISETPICKAPERTUREPOSITION	0xf5	GpiSetPick-AperturePosition
API_GPISETPICKAPERTURESIZE	0xf6	GpiSetPickApertureSize
API_GPISETREGION	0xf7	GpiSetRegion
API_GPISETSEGMENTATTRS	0xf8	GpiSetSegmentAttrs
API_GPISETSEGMENTPRIORITY	0xf9	GpiSetSegmentPriority
API_GPISETSEGMENTTRANSFORMMATRIX	0xfa	GpiSetSegment-TransformMatrix
API_GPISETSTOPDRAW	0xfb	GpiSetStopDraw
API_GPISETTAG	0xfc	GpiSetTag
API_GPISETTEXTALIGNMENT	0xfd	GpiSetTextAlignment
API_GPISETVIEWINGLIMITS	0xfe	GpiSetViewingLimits

Symbol	Index	API
API_GPISETVIEWINGTRANSFORMMATRIX	0xff	GpiSetViewing-TransformMatrix
API_GPISTROKEPATH	0x100	GpiStrokePath
API_GPITRANSLATE	0x101	GpiTranslate
API_GPIUNLOADFONTS	0x102	GpiUnloadFonts
API_GPIUNLOADPUBLICFONTS	0x103	GpiUnloadPublicFonts
API_GPIWCBITBLT	0x104	GpiWCBitBlt

Pen for OS/2 Calls

Symbol	Index	API
API_REDDEREGISTERRECOCOMMAND	0x0	RedDeregisterReco-Command
API_REDQUERYRECOCOMMAND	0x1	RedQueryReco-Command
API_REDQUERYRECOHANDLE	0x2	RedQueryRecoHandle
API_REDQUERYRECOSUBSYSTEM	0x3	RedQueryReco-Subsystem
API_REDREADOBJECTEVENTMAP	0x4	RedReadObject-EventMap
API_REDRECODATAFROMENV	0x5	RedRecoDataFromEnv
API_REDRECOIDFROMNAME	0x6	RedRecoIDFromName
API_REDRECONAMEFROMID	0x7	RedRecoNameFromID
API_REDREGISTERRECOCOMMAND	0x8	RedRegisterReco-Command
API_REDWRITEOBJECTEVENTMAP	0x9	RedWriteObject-EventMap
API_VKPCLOSEKB	0xa	VkpCloseKb
API_VKPDELETEKEYBOARD	0xb	VkpDeleteKeyboard
API_VKPHIDEKEYBOARD	0xc	VkpHideKeyboard
API_VKPISKBHIDDEN	0xd	VkpIsKbHidden
API_VKPISKBRUNNING	0xe	VkpIsKbRunning
API_VKPLOADKEYBOARD	0xf	VkpLoadKeyboard
API_VKPQUERYKBPOS	0x10	VkpQueryKbPos
API_VKPSETKBPOS	0x11	VkpSetKbPos
API_WRTCONTROLDISPLAYBACKLIGHT	0x12	WrtControlDisplay-Backlight
API_WRTENUMINPUTDRIVERS	0x13	WrtEnumInputDrivers
API_WRTMAPPOINTLONG	0x14	WrtMapPointLong
API_WRTQUERYBUTTONCAPS	0x15	WrtQueryButtonCaps
API_WRTQUERYDISPLAYCAPS	0x16	WrtQueryDisplayCaps
API_WRTQUERYEVENTDATA	0x17	WrtQueryEventData
API_WRTQUERYINPUTDEVICENAMES	0x18	WrtQueryInput-DeviceNames
API_WRTQUERYINPUTDEVICEVARIABLE	0x19	WrtQueryInput-DeviceVariable
API_WRTQUERYLOCATORCAPS	0x1a	WrtQueryLocatorCaps
API_WRTQUERYPOINTAUXDATA	0x1b	WrtQueryPointAuxData
API_WRTQUERYSTROKEDATA	0x1c	WrtQueryStrokeData
API_WRTQUERYSYSVALUE	0x1d	WrtQuerySysValue
API_WRTQUERYSYSTEMCAPS	0x1e	WrtQuerySystemCaps

Symbol	Index	API
API_WRTREADSYSVALUE	0x1f	WrtReadSysValue
API_WRTSETINPUTDEVICEVARIABLE	0x20	WrtSetInputDevice-Variable
API_WRTSETSTRICTEMULATION	0x21	WrtSetStrictEmulation
API_WRTSETSYSVALUE	0x22	WrtSetSysValue
API_WRTWAITACTIVE	0x23	WrtWaitActive
API_WRTWRITESYSVALUE	0x24	WrtWriteSysValue

Pic* Calls

Symbol	Index	API
API_PICICHG	0x0	PicIchg
API_PICPRINT	0x1	PicPrint

Prf* Calls

Symbol	Index	API
API_PRFADDPROGRAM	0x0	PrfAddProgram
API_PRFCHANGEPROGRAM	0x1	PrfChangeProgram
API_PRFCLOSEPROFILE	0x2	PrfCloseProfile
API_PRFDESTROYGROUP	0x3	PrfDestroyGroup
API_PRFOPENPROFILE	0x4	PrfOpenProfile
API_PRFQUERYDEFINITION	0x5	PrfQueryDefinition
API_PRFQUERYPROFILE	0x6	PrfQueryProfile
API_PRFQUERYPROFILEDATA	0x7	PrfQueryProfileData
API_PRFQUERYPROFILEINT	0x8	PrfQueryProfileInt
API_PRFQUERYPROFILESIZE	0x9	PrfQueryProfileSize
API_PRFQUERYPROFILESTRING	0xa	PrfQueryProfileString
API_PRFQUERYPROGRAMTITLES	0xb	PrfQueryProgramTitles
API_PRFREMOVEPROGRAM	0xc	PrfRemoveProgram
API_PRFRESET	0xd	PrfReset
API_PRFWRITEPROFILEDATA	0xe	PrfWriteProfileData
API_PRFWRITEPROFILESTRING	0xf	PrfWriteProfileString

Prt* Calls

Symbol	Index	API
API_PRTABORT	0x0	PrtAbort
API_PRTCLOSE	0x1	PrtClose
API_PRTDEVIOCTL	0x2	PrtDevIOCtl
API_PRTOPEN	0x3	PrtOpen
API_PRTWRITE	0x4	PrtWrite

Spl* Calls

Symbol	Index	API
API_SPLCONTROLDEVICE	0x0	SplControlDevice
API_SPLCOPYJOB	0x1	SplCopyJob
API_SPLCREATEDEVICE	0x2	SplCreateDevice

Symbol	Index	API
API_SPLCREATEQUEUE	0x3	SplCreateQueue
API_SPLDELETEDEVICE	0x4	SplDeleteDevice
API_SPLDELETEJOB	0x5	SplDeleteJob
API_SPLDELETEQUEUE	0x6	SplDeleteQueue
API_SPLENUMDEVICE	0x7	SplEnumDevice
API_SPLENUMDRIVER	0x8	SplEnumDriver
API_SPLENUMJOB	0x9	SplEnumJob
API_SPLENUMPORT	0xa	SplEnumPort
API_SPLENUMPRINTER	0xb	SplEnumPrinter
API_SPLENUMQUEUE	0xc	SplEnumQueue
API_SPLENUMQUEUEPROCESSOR	0xd	SplEnumQueue-Processor
API_SPLHOLDJOB	0xe	SplHoldJob
API_SPLHOLDQUEUE	0xf	SplHoldQueue
API_SPLMESSAGEBOX	0x10	SplMessageBox
API_SPLPURGEQUEUE	0x11	SplPurgeQueue
API_SPLQMABORT	0x12	SplQmAbort
API_SPLQMABORTDOC	0x13	SplQmAbortDoc
API_SPLQMCLOSE	0x14	SplQmClose
API_SPLQMENDDOC	0x15	SplQmEndDoc
API_SPLQMOPEN	0x16	SplQmOpen
API_SPLQMSTARTDOC	0x17	SplQmStartDoc
API_SPLQMWRITE	0x18	SplQmWrite
API_SPLQUERYDEVICE	0x19	SplQueryDevice
API_SPLQUERYJOB	0x1a	SplQueryJob
API_SPLQUERYQUEUE	0x1b	SplQueryQueue
API_SPLRELEASEJOB	0x1c	SplReleaseJob
API_SPLRELEASEQUEUE	0x1d	SplReleaseQueue
API_SPLSETDEVICE	0x1e	SplSetDevice
API_SPLSETJOB	0x1f	SplSetJob
API_SPLSETQUEUE	0x20	SplSetQueue
API_SPLSTDCLOSE	0x21	SplStdClose
API_SPLSTDDELETE	0x22	SplStdDelete
API_SPLSTDGETBITS	0x23	SplStdGetBits
API_SPLSTDOPEN	0x24	SplStdOpen
API_SPLSTDQUERYLENGTH	0x25	SplStdQueryLength
API_SPLSTDSTART	0x26	SplStdStart
API_SPLSTDSTOP	0x27	SplStdStop

Win* Calls

Symbol	Index	API
API_WINADDATOM	0x0	WinAddAtom
API_WINADDSWITCHENTRY	0x1	WinAddSwitchEntry
API_WINALARM	0x2	WinAlarm
API_WINASSOCIATEHELPINSTANCE	0x3	WinAssociateHelp-Instance
API_WINBEGINENUMWINDOWS	0x4	WinBeginEnum-Windows
API_WINBEGINPAINT	0x5	WinBeginPaint
API_WINBROADCASTMSG	0x6	WinBroadcastMsg

Symbol	Index	API
API_WINCALCFRAMERECT	0x7	WinCalcFrameRect
API_WINCALLMSGFILTER	0x8	WinCallMsgFilter
API_WINCANCELSHUTDOWN	0x9	WinCancelShutdown
API_WINCHANGESWITCHENTRY	0xa	WinChangeSwitchEntry
API_WINCLOSECLIPBRD	0xb	WinCloseClipbrd
API_WINCOMPARESTRINGS	0xc	WinCompareStrings
API_WINCOPYACCELTABLE	0xd	WinCopyAccelTable
API_WINCOPYRECT	0xe	WinCopyRect
API_WINCPTRANSLATECHAR	0xf	WinCpTranslateChar
API_WINCPTRANSLATESTRING	0x10	WinCpTranslateString
API_WINCREATEACCELTABLE	0x11	WinCreateAccelTable
API_WINCREATEATOMTABLE	0x12	WinCreateAtomTable
API_WINCREATECURSOR	0x13	WinCreateCursor
API_WINCREATEDLG	0x14	WinCreateDlg
API_WINCREATEFRAMECONTROLS	0x15	WinCreateFrame-Controls
API_WINCREATEHELPINSTANCE	0x16	WinCreateHelpInstance
API_WINCREATEHELPTABLE	0x17	WinCreateHelpTable
API_WINCREATEMENU	0x18	WinCreateMenu
API_WINCREATEMSGQUEUE	0x19	WinCreateMsgQueue
API_WINCREATEOBJECT	0x1a	WinCreateObject
API_WINCREATEPOINTER	0x1b	WinCreatePointer
API_WINCREATEPOINTERINDIRECT	0x1c	WinCreatePointer-Indirect
API_WINCREATESTDWINDOW	0x1d	WinCreateStdWindow
API_WINCREATESWITCHENTRY	0x1e	WinCreateSwitchEntry
API_WINCREATEWINDOW	0x1f	WinCreateWindow
API_WINDDEINITIATE	0x20	WinDdeInitiate
API_WINDDEPOSTMSG	0x21	WinDdePostMsg
API_WINDDERESPOND	0x22	WinDdeRespond
API_WINDEFDLGPROC	0x23	WinDefDlgProc
API_WINDEFFILEDLGPROC	0x24	WinDefFileDlgProc
API_WINDEFFONTDLGPROC	0x25	WinDefFontDlgProc
API_WINDEFWINDOWPROC	0x26	WinDefWindowProc
API_WINDELETEATOM	0x27	WinDeleteAtom
API_WINDELETELIBRARY	0x28	WinDeleteLibrary
API_WINDELETEPROCEDURE	0x29	WinDeleteProcedure
API_WINDEREGISTEROBJECTCLASS	0x2a	WinDeregister-ObjectClass
API_WINDESTROYACCELTABLE	0x2b	WinDestroyAccelTable
API_WINDESTROYATOMTABLE	0x2c	WinDestroyAtomTable
API_WINDESTROYCURSOR	0x2d	WinDestroyCursor
API_WINDESTROYHELPINSTANCE	0x2e	WinDestroyHelp-Instance
API_WINDESTROYMSGQUEUE	0x2f	WinDestroyMsgQueue
API_WINDESTROYOBJECT	0x30	WinDestroyObject
API_WINDESTROYPOINTER	0x31	WinDestroyPointer
API_WINDESTROYWINDOW	0x32	WinDestroyWindow
API_WINDISMISSDLG	0x33	WinDismissDlg
API_WINDISPATCHMSG	0x34	WinDispatchMsg
API_WINDLGBOX	0x35	WinDlgBox

Symbol	Index	API
API_WINDRAWBITMAP	0x36	WinDrawBitmap
API_WINDRAWBORDER	0x37	WinDrawBorder
API_WINDRAWPOINTER	0x38	WinDrawPointer
API_WINDRAWTEXT	0x39	WinDrawText
API_WINEMPTYCLIPBRD	0x3a	WinEmptyClipbrd
API_WINENABLEPHYSINPUT	0x3b	WinEnablePhysInput
API_WINENABLEWINDOW	0x3c	WinEnableWindow
API_WINENABLEWINDOWUPDATE	0x3d	WinEnableWindow-Update
API_WINENDENUMWINDOWS	0x3e	WinEndEnumWindows
API_WINENDPAINT	0x3f	WinEndPaint
API_WINENUMCLIPBRDFMTS	0x40	WinEnumClipbrdFmts
API_WINENUMDLGITEM	0x41	WinEnumDlgItem
API_WINENUMOBJECTCLASSES	0x42	WinEnumObjectClasses
API_WINEQUALRECT	0x43	WinEqualRect
API_WINEXCLUDEUPDATEREGION	0x44	WinExcludeUpdate-Region
API_WINFILEDLG	0x45	WinFileDlg
API_WINFILLRECT	0x46	WinFillRect
API_WINFINDATOM	0x47	WinFindAtom
API_WINFLASHWINDOW	0x48	WinFlashWindow
API_WINFOCUSCHANGE	0x49	WinFocusChange
API_WINFONTDLG	0x4a	WinFontDlg
API_WINFREEERRORINFO	0x4b	WinFreeErrorInfo
API_WINFREEFILEDLGLIST	0x4c	WinFreeFileDlgList
API_WINFREEFILEICON	0x4d	WinFreeFileIcon
API_WINGETCLIPPS	0x4e	WinGetClipPS
API_WINGETCURRENTTIME	0x4f	WinGetCurrentTime
API_WINGETDLGMSG	0x50	WinGetDlgMsg
API_WINGETERRORINFO	0x51	WinGetErrorInfo
API_WINGETKEYSTATE	0x52	WinGetKeyState
API_WINGETLASTERROR	0x53	WinGetLastError
API_WINGETMAXPOSITION	0x54	WinGetMaxPosition
API_WINGETMINPOSITION	0x55	WinGetMinPosition
API_WINGETMSG	0x56	WinGetMsg
API_WINGETNEXTWINDOW	0x57	WinGetNextWindow
API_WINGETPS	0x58	WinGetPS
API_WINGETPHYSKEYSTATE	0x59	WinGetPhysKeyState
API_WINGETSCREENPS	0x5a	WinGetScreenPS
API_WINGETSYSBITMAP	0x5b	WinGetSysBitmap
API_WININSENDMSG	0x5c	WinInSendMsg
API_WININFLATERECT	0x5d	WinInflateRect
API_WININITIALIZE	0x5e	WinInitialize
API_WININTERSECTRECT	0x5f	WinIntersectRect
API_WININVALIDATERECT	0x60	WinInvalidateRect
API_WININVALIDATEREGION	0x61	WinInvalidateRegion
API_WININVERTRECT	0x62	WinInvertRect
API_WINISCHILD	0x63	WinIsChild
API_WINISPHYSINPUTENABLED	0x64	WinIsPhysInputEnabled
API_WINISRECTEMPTY	0x65	WinIsRectEmpty
API_WINISTHREADACTIVE	0x66	WinIsThreadActive

Symbol	Index	API
API_WINISWINDOW	0x67	WinIsWindow
API_WINISWINDOWENABLED	0x68	WinIsWindowEnabled
API_WINISWINDOWSHOWING	0x69	WinIsWindowShowing
API_WINISWINDOWVISIBLE	0x6a	WinIsWindowVisible
API_WINLOADACCELTABLE	0x6b	WinLoadAccelTable
API_WINLOADDLG	0x6c	WinLoadDlg
API_WINLOADFILEICON	0x6d	WinLoadFileIcon
API_WINLOADHELPTABLE	0x6e	WinLoadHelpTable
API_WINLOADLIBRARY	0x6f	WinLoadLibrary
API_WINLOADMENU	0x70	WinLoadMenu
API_WINLOADMESSAGE	0x71	WinLoadMessage
API_WINLOADPOINTER	0x72	WinLoadPointer
API_WINLOADPROCEDURE	0x73	WinLoadProcedure
API_WINLOADSTRING	0x74	WinLoadString
API_WINLOCKVISREGIONS	0x75	WinLockVisRegions
API_WINLOCKWINDOWUPDATE	0x76	WinLockWindow-Update
API_WINMAKEPOINTS	0x77	WinMakePoints
API_WINMAKERECT	0x78	WinMakeRect
API_WINMAPDLGPOINTS	0x79	WinMapDlgPoints
API_WINMAPWINDOWPOINTS	0x7a	WinMapWindowPoints
API_WINMESSAGEBOX	0x7b	WinMessageBox
API_WINMULTWINDOWFROMIDS	0x7c	WinMultWindow-FromIDs
API_WINNEXTCHAR	0x7d	WinNextChar
API_WINOFFSETRECT	0x7e	WinOffsetRect
API_WINOPENCLIPBRD	0x7f	WinOpenClipbrd
API_WINOPENWINDOWDC	0x80	WinOpenWindowDC
API_WINPEEKMSG	0x81	WinPeekMsg
API_WINPOPUPMENU	0x82	WinPopupMenu
API_WINPOSTMSG	0x83	WinPostMsg
API_WINPOSTQUEUEMSG	0x84	WinPostQueueMsg
API_WINPREVCHAR	0x85	WinPrevChar
API_WINPROCESSDLG	0x86	WinProcessDlg
API_WINPTINRECT	0x87	WinPtInRect
API_WINQUERYACCELTABLE	0x88	WinQueryAccelTable
API_WINQUERYACTIVEWINDOW	0x89	WinQueryActive-Window
API_WINQUERYANCHORBLOCK	0x8a	WinQueryAnchorBlock
API_WINQUERYATOMLENGTH	0x8b	WinQueryAtomLength
API_WINQUERYATOMNAME	0x8c	WinQueryAtomName
API_WINQUERYATOMUSAGE	0x8d	WinQueryAtomUsage
API_WINQUERYCAPTURE	0x8e	WinQueryCapture
API_WINQUERYCLASSINFO	0x8f	WinQueryClassInfo
API_WINQUERYCLASSNAME	0x90	WinQueryClassName
API_WINQUERYCLASSTHUNKPROC	0x91	WinQueryClass-ThunkProc
API_WINQUERYCLIPBRDDATA	0x92	WinQueryClipbrdData
API_WINQUERYCLIPBRDFMTINFO	0x93	WinQueryClipbrd-FmtInfo
API_WINQUERYCLIPBRDOWNER	0x94	WinQueryClipbrdOwner

Symbol	Index	API
API_WINQUERYCLIPBRDVIEWER	0x95	WinQueryClipbrd-Viewer
API_WINQUERYCP	0x96	WinQueryCp
API_WINQUERYCPLIST	0x97	WinQueryCpList
API_WINQUERYCURSORINFO	0x98	WinQueryCursorInfo
API_WINQUERYDESKTOPBKGND	0x99	WinQueryDesktop-Bkgnd
API_WINQUERYDESKTOPWINDOW	0x9a	WinQueryDesktop-Window
API_WINQUERYDLGITEMSHORT	0x9b	WinQueryDlgItemShort
API_WINQUERYDLGITEMTEXT	0x9c	WinQueryDlgItemText
API_WINQUERYDLGITEMTEXTLENGTH	0x9d	WinQueryDlgItem-TextLength
API_WINQUERYFOCUS	0x9e	WinQueryFocus
API_WINQUERYHELPINSTANCE	0x9f	WinQueryHelpInstance
API_WINQUERYMSGPOS	0xa0	WinQueryMsgPos
API_WINQUERYMSGTIME	0xa1	WinQueryMsgTime
API_WINQUERYOBJECT	0xa2	WinQueryObject
API_WINQUERYOBJECTWINDOW	0xa3	WinQueryObject-Window
API_WINQUERYPOINTER	0xa4	WinQueryPointer
API_WINQUERYPOINTERINFO	0xa5	WinQueryPointerInfo
API_WINQUERYPOINTERPOS	0xa6	WinQueryPointerPos
API_WINQUERYPRESPARAM	0xa7	WinQueryPresParam
API_WINQUERYQUEUEINFO	0xa8	WinQueryQueueInfo
API_WINQUERYQUEUESTATUS	0xa9	WinQueryQueueStatus
API_WINQUERYSESSIONTITLE	0xaa	WinQuerySessionTitle
API_WINQUERYSWITCHENTRY	0xab	WinQuerySwitchEntry
API_WINQUERYSWITCHHANDLE	0xac	WinQuerySwitchHandle
API_WINQUERYSWITCHLIST	0xad	WinQuerySwitchList
API_WINQUERYSYSCOLOR	0xae	WinQuerySysColor
API_WINQUERYSYSMODALWINDOW	0xaf	WinQuerySysModal-Window
API_WINQUERYSYSPOINTER	0xb0	WinQuerySysPointer
API_WINQUERYSYSVALUE	0xb1	WinQuerySysValue
API_WINQUERYSYSTEMATOMTABLE	0xb2	WinQuerySystem-AtomTable
API_WINQUERYTASKSIZEPOS	0xb3	WinQueryTaskSizePos
API_WINQUERYTASKTITLE	0xb4	WinQueryTaskTitle
API_WINQUERYUPDATERECT	0xb5	WinQueryUpdateRect
API_WINQUERYUPDATEREGION	0xb6	WinQueryUpdateRegion
API_WINQUERYVERSION	0xb7	WinQueryVersion
API_WINQUERYWINDOW	0xb8	WinQueryWindow
API_WINQUERYWINDOWDC	0xb9	WinQueryWindowDC
API_WINQUERYWINDOWMODEL	0xba	WinQueryWindow-Model
API_WINQUERYWINDOWPOS	0xbb	WinQueryWindowPos
API_WINQUERYWINDOWPROCESS	0xbc	WinQueryWindow-Process
API_WINQUERYWINDOWPTR	0xbd	WinQueryWindowPtr
API_WINQUERYWINDOWRECT	0xbe	WinQueryWindowRect

Symbol	Index	API
API_WINQUERYWINDOWTEXT	0xbf	WinQueryWindowText
API_WINQUERYWINDOWTEXTLENGTH	0xc0	WinQueryWindow-TextLength
API_WINQUERYWINDOWTHUNKPROC	0xc1	WinQueryWindow-ThunkProc
API_WINQUERYWINDOWULONG	0xc2	WinQueryWindow-ULong
API_WINQUERYWINDOWUSHORT	0xc3	WinQueryWindow-UShort
API_WINREALIZEPALETTE	0xc4	WinRealizePalette
API_WINREGISTERCLASS	0xc5	WinRegisterClass
API_WINREGISTEROBJECTCLASS	0xc6	WinRegisterObjectClass
API_WINREGISTERUSERDATATYPE	0xc7	WinRegisterUser-Datatype
API_WINREGISTERUSERMSG	0xc8	WinRegisterUserMsg
API_WINRELEASEHOOK	0xc9	WinReleaseHook
API_WINRELEASEPS	0xca	WinReleasePS
API_WINREMOVEPRESPARAM	0xcb	WinRemovePresParam
API_WINREMOVESWITCHENTRY	0xcc	WinRemoveSwitchEntry
API_WINREPLACEOBJECTCLASS	0xcd	WinReplaceObjectClass
API_WINREQUESTMUTEXSEM	0xce	WinRequestMutexSem
API_WINRESTOREWINDOWPOS	0xcf	WinRestoreWindowPos
API_WINSAVEWINDOWPOS	0xd0	WinSaveWindowPos
API_WINSCROLLWINDOW	0xd1	WinScrollWindow
API_WINSENDDLGITEMMSG	0xd2	WinSendDlgItemMsg
API_WINSENDMSG	0xd3	WinSendMsg
API_WINSETACCELTABLE	0xd4	WinSetAccelTable
API_WINSETACTIVEWINDOW	0xd5	WinSetActiveWindow
API_WINSETCAPTURE	0xd6	WinSetCapture
API_WINSETCLASSMSGINTEREST	0xd7	WinSetClassMsgInterest
API_WINSETCLASSTHUNKPROC	0xd8	WinSetClassThunkProc
API_WINSETCLIPBRDDATA	0xd9	WinSetClipbrdData
API_WINSETCLIPBRDOWNER	0xda	WinSetClipbrdOwner
API_WINSETCLIPBRDVIEWER	0xdb	WinSetClipbrdViewer
API_WINSETCP	0xdc	WinSetCp
API_WINSETDESKTOPBKGND	0xdd	WinSetDesktopBkgnd
API_WINSETDLGITEMSHORT	0xde	WinSetDlgItemShort
API_WINSETDLGITEMTEXT	0xdf	WinSetDlgItemText
API_WINSETFILEICON	0xe0	WinSetFileIcon
API_WINSETFOCUS	0xe1	WinSetFocus
API_WINSETHOOK	0xe2	WinSetHook
API_WINSETKEYBOARDSTATETABLE	0xe3	WinSetKeyboard-StateTable
API_WINSETMSGINTEREST	0xe4	WinSetMsgInterest
API_WINSETMSGMODE	0xe5	WinSetMsgMode
API_WINSETMULTWINDOWPOS	0xe6	WinSetMultWindowPos
API_WINSETOBJECTDATA	0xe7	WinSetObjectData
API_WINSETOWNER	0xe8	WinSetOwner
API_WINSETPARENT	0xe9	WinSetParent
API_WINSETPOINTER	0xea	WinSetPointer
API_WINSETPOINTEROWNER	0xeb	WinSetPointerOwner

Symbol	Index	API
API_WINSETPOINTERPOS	0xec	WinSetPointerPos
API_WINSETPRESPARAM	0xed	WinSetPresParam
API_WINSETRECT	0xee	WinSetRect
API_WINSETRECTEMPTY	0xef	WinSetRectEmpty
API_WINSETSYNCHROMODE	0xf0	WinSetSynchroMode
API_WINSETSYSCOLORS	0xf1	WinSetSysColors
API_WINSETSYSMODALWINDOW	0xf2	WinSetSysModal-Window
API_WINSETSYSVALUE	0xf3	WinSetSysValue
API_WINSETWINDOWBITS	0xf4	WinSetWindowBits
API_WINSETWINDOWPOS	0xf5	WinSetWindowPos
API_WINSETWINDOWPTR	0xf6	WinSetWindowPtr
API_WINSETWINDOWTEXT	0xf7	WinSetWindowText
API_WINSETWINDOWTHUNKPROC	0xf8	WinSetWindow-ThunkProc
API_WINSETWINDOWULONG	0xf9	WinSetWindowULong
API_WINSETWINDOWUSHORT	0xfa	WinSetWindowUShort
API_WINSHOWCURSOR	0xfb	WinShowCursor
API_WINSHOWPOINTER	0xfc	WinShowPointer
API_WINSHOWTRACKRECT	0xfd	WinShowTrackRect
API_WINSHOWWINDOW	0xfe	WinShowWindow
API_WINSHUTDOWNSYSTEM	0xff	WinShutdownSystem
API_WINSTARTAPP	0x100	WinStartApp
API_WINSTARTTIMER	0x101	WinStartTimer
API_WINSTOPTIMER	0x102	WinStopTimer
API_WINSTOREWINDOWPOS	0x103	WinStoreWindowPos
API_WINSUBCLASSWINDOW	0x104	WinSubclassWindow
API_WINSUBSTITUTESTRINGS	0x105	WinSubstituteStrings
API_WINSUBTRACTRECT	0x106	WinSubtractRect
API_WINSWITCHTOPROGRAM	0x107	WinSwitchToProgram
API_WINTERMINATE	0x108	WinTerminate
API_WINTERMINATEAPP	0x109	WinTerminateApp
API_WINTRACKRECT	0x10a	WinTrackRect
API_WINTRANSLATEACCEL	0x10b	WinTranslateAccel
API_WINUNIONRECT	0x10c	WinUnionRect
API_WINUPDATEWINDOW	0x10d	WinUpdateWindow
API_WINUPPER	0x10e	WinUpper
API_WINUPPERCHAR	0x10f	WinUpperChar
API_WINVALIDATERECT	0x110	WinValidateRect
API_WINVALIDATEREGION	0x111	WinValidateRegion
API_WINWAITEVENTSEM	0x112	WinWaitEventSem
API_WINWAITMSG	0x113	WinWaitMsg
API_WINWAITMUXWAITSEM	0x114	WinWaitMuxWaitSem
API_WINWINDOWFROMDC	0x115	WinWindowFromDC
API_WINWINDOWFROMID	0x116	WinWindowFromID
API_WINWINDOWFROMPOINT	0x117	WinWindowFromPoint

OS/2 2.1 Win Calls*

Symbol	Index	API
API_WINCHECKINPUT	0x118	WinCheckInput
API_WINLOCKPOINTERUPDATE	0x119	WinLockPointerUpdate
API_WINLOCKUPSYSTEM	0x11a	WinLockupSystem
API_WINQUERYSYSPOINTERDATA	0x11b	WinQuerySys-PointerData
API_WINSETPOINTEROWNER	0x11c	WinSetPointerOwner
API_WINSETSYSPOINTERDATA	0x11d	WinSetSysPointerData
API_WINUNLOCKSYSTEM	0x11e	WinUnlockSystem

MMPM/2 Calls

Symbol	Index	API
API_SPIASSOCIATE	0x0	SpiAssociate
API_SPICREATESTREAM	0x1	SpiCreateStream
API_SPIDESTROYSTREAM	0x2	SpiDestroyStream
API_SPIDETERMINESYNCMASTER	0x3	SpiDetermineSync-Master
API_SPIDISABLEEVENT	0x4	SpiDisableEvent
API_SPIDISABLESYNC	0x5	SpiDisableSync
API_SPIENABLEEVENT	0x6	SpiEnableEvent
API_SPIENABLESYNC	0x7	SpiEnableSync
API_SPIENUMERATEHANDLERS	0x8	SpiEnumerateHandlers
API_SPIENUMERATEPROTOCOLS	0x9	SpiEnumerateProtocols
API_SPIGETHANDLER	0xa	SpiGetHandler
API_SPIGETPROTOCOL	0xb	SpiGetProtocol
API_SPIGETTIME	0xc	SpiGetTime
API_SPIINSTALLPROTOCOL	0xd	SpiInstallProtocol
API_SPISEEKSTREAM	0xe	SpiSeekStream
API_SPISENDMSG	0xf	SpiSendMsg
API_SPISTARTSTREAM	0x10	SpiStartStream
API_SPISTOPSTREAM	0x11	SpiStopStream
API_MCIDELETEGROUP	0x12	mciDeleteGroup
API_MCIGETDEVICEID	0x13	mciGetDeviceID
API_MCIGETERRORSTRING	0x14	mciGetErrorString
API_MCIMAKEGROUP	0x15	mciMakeGroup
API_MCIPLAYFILE	0x16	mciPlayFile
API_MCIPLAYRESOURCE	0x17	mciPlayResource
API_MCIQUERYSYSVALUE	0x18	mciQuerySysValue
API_MCIRECORDAUDIOFILE	0x19	mciRecordAudioFile
API_MCISENDCOMMAND	0x1a	mciSendCommand
API_MCISENDSTRING	0x1b	mciSendString
API_MCISETSYSVALUE	0x1c	mciSetSysValue
API_MDMDRIVERNOTIFY	0x1d	mdmDriverNotify
API_MMIOADVANCE	0x1e	mmioAdvance
API_MMIOASCEND	0x1f	mmioAscend
API_MMIOCFADDELEMENT	0x20	mmioCFAddElement
API_MMIOCFADDENTRY	0x21	mmioCFAddEntry
API_MMIOCFCHANGEENTRY	0x22	mmioCFChangeEntry
API_MMIOCFCLOSE	0x23	mmioCFClose

Symbol	Index	API
API_MMIOCFCOMPACT	0x24	mmioCFCompact
API_MMIOCFCOPY	0x25	mmioCFCopy
API_MMIOCFDELETEENTRY	0x26	mmioCFDeleteEntry
API_MMIOCFFINDENTRY	0x27	mmioCFFindEntry
API_MMIOCFGETINFO	0x28	mmioCFGetInfo
API_MMIOCFOPEN	0x29	mmioCFOpen
API_MMIOCFSETINFO	0x2a	mmioCFSetInfo
API_MMIOCLOSE	0x2b	mmioClose
API_MMIOCREATECHUNK	0x2c	mmioCreateChunk
API_MMIODESCEND	0x2d	mmioDescend
API_MMIODETERMINESSIOPROC	0x2e	mmioDetermine-SSIOProc
API_MMIOFINDELEMENT	0x2f	mmioFindElement
API_MMIOFLUSH	0x30	mmioFlush
API_MMIOGETFORMATNAME	0x31	mmioGetFormatName
API_MMIOGETFORMATS	0x32	mmioGetFormats
API_MMIOGETHEADER	0x33	mmioGetHeader
API_MMIOGETINFO	0x34	mmioGetInfo
API_MMIOGETLASTERROR	0x35	mmioGetLastError
API_MMIOIDENTIFYFILE	0x36	mmioIdentifyFile
API_MMIOIDENTIFYSTORAGESYSTEM	0x37	mmioIdentifyStorage-System
API_MMIOINIFILECODEC	0x38	mmioIniFileCODEC
API_MMIOINIFILEHANDLER	0x39	mmioIniFileHandler
API_MMIOINSTALLIOPROC	0x3a	mmioInstallIOProc
API_MMIOLOADCODECPROC	0x3b	mmioLoadCODECProc
API_MMIOOPEN	0x3c	mmioOpen
API_MMIOQUERYCODECNAME	0x3d	mmioQueryCODEC-Name
API_MMIOQUERYCODECNAMELENGTH	0x3e	mmioQueryCODEC-NameLength
API_MMIOQUERYFORMATCOUNT	0x3f	mmioQueryFormat-Count
API_MMIOQUERYHEADERLENGTH	0x40	mmioQueryHeader-Length
API_MMIOQUERYIOPROCMODULEHANDLE	0x41	mmioQueryIOProc-ModuleHandle
API_MMIOREAD	0x42	mmioRead
API_MMIOREMOVEELEMENT	0x43	mmioRemoveElement
API_MMIOSEEK	0x44	mmioSeek
API_MMIOSENDMESSAGE	0x45	mmioSendMessage
API_MMIOSET	0x46	mmioSet
API_MMIOSETBUFFER	0x47	mmioSetBuffer
API_MMIOSETHEADER	0x48	mmioSetHeader
API_MMIOSETINFO	0x49	mmioSetInfo
API_MMIOSTRINGTOFOURCC	0x4a	mmioStringToFOURCC
API_MMIOWRITE	0x4b	mmioWrite
API_WINCREATESECONDARYWINDOW	0x4c	WinCreateSecondary-Window

Symbol	Index	API
API_WINDEFSECONDARYWINDOWPROC	0x4d	WinDefSecondary-WindowProc
API_WINDEFAULTSIZE	0x4e	WinDefaultSize
API_WINDESTROYSECONDARYWINDOW	0x4f	WinDestroy-SecondaryWindow
API_WINDISMISSSECONDARYWINDOW	0x50	WinDismiss-SecondaryWindow
API_WININSERTDEFAULTSIZE	0x51	WinInsertDefaultSize
API_WINLOADSECONDARYWINDOW	0x52	WinLoadSecondary-Window
API_WINPROCESSSECONDARYWINDOW	0x53	WinProcess-SecondaryWindow
API_WINQUERYSECONDARYHWND	0x54	WinQuerySecondary-HWND
API_WINREGISTERCIRCULARSLIDER	0x55	WinRegisterCircular-Slider
API_WINREGISTERGRAPHICBUTTON	0x56	WinRegisterGraphic-Button
API_WINSECONDARYMESSAGEBOX	0x57	WinSecondary-MessageBox
API_WINSECONDARYWINDOW	0x58	WinSecondaryWindow

⇓ **ULONG** *ulError*
OS/2 error value that is to be checked for. If the error is encountered within the application and the *ulReturn* parameter value matches the actual API return, the error will not be passed to ***ViewPort*** or written to the logging files.

⇓ **ULONG** *ulReturn*
API return value. If the error *ulError* is encountered within the application from the API, the *ulReturn* value is checked to make sure that the desired error is prevented from being passed to ***ViewPort*** or written to the logging files.

Return Value

A return value of APIERR_NONE (0) indicates successful registering of API for error filtering otherwise the following error values will be returned:

Symbol	Value	Problem
APIERR_INVALID_API	1	Invalid API index used.
APIERR_INVALID_API_FAMILY	2	Invalid API family used.
APIERR_NO_MEM_AVAIL	3	No memory available to register API.
APIERR_INVALID_HVAL	4	Invalid handle.

Comments

When you encounter an API that due to its design returns frequent error codes, you may decide that you want to prevent this message from being sent to the log files or to ***ViewPort***. A good example is the **DrgFreeDraginfo** API which when you are dragging an object will

return 0 indicating an error. The error is PMERR_SOURCE_SAME_AS_TARGET where the object you are dragging is being dragged through its owner window which in this case cannot accept the drop.

Similar such errors occur when you are in other modal-like operations. Depending on the scenario, you may get thousands of error returns while performing the operation and this can cause an overload condition to occur where the system will have seemed to have hung but in reality, it is trying to process the messages that have been generated.

The best method of determining when you have this kind of condition is when you perform an operation and it seems sluggish. When you have completed the operation, like releasing the mouse button in a drag operation, a flurry of error messages will occur within the corresponding ***ViewPort*** error window. Even though ***ViewPort*** provides error filtering, this does not relieve the message flow from the application that you are performing API validation on and since ***ViewPort*** will have still received the error message before it was filtered.

Example

When using the **ValFilterErr** function, it is best to first determine the full error return. You can either use the value returned through the API by using the debugging or some other method. Alternately, you can view the full value within ***ViewPort***.

It should be remembered that when you are using ***Validator***, parameter errors include additional error information. This must be included with the **ValFilterErr**. Also, when dealing with OS/2 Presentation Manager errors that are retrieved through **WinGetLastError**, not only is an error value returned (PMERR_*) but also a severity is returned. This severity is returned in bits 16 - 24. A macro, **ERRORIDERROR(errid)**, is used to determine the PMERR_* value. A second macro, **ERRORIDSEV(errid)**, is used to determine the severity value.

The following example shows how to use the function with a OS/2 Presentation Manager for a parameter return error for **GpiBitBlt**:

```
ValFilterErr(hval, APIFAMILY_GPI, API_GPIBITBLT, 0x01041003UL, 0UL);
```

The error value is comprised of:

```
PERR_GBB03_COUNTLT3
SEVERITY_WARNING
PMERR_PARAMETER_OUT_OF_RANGE
```

Another example is for a non-parameter error for **DosPostEventSem**:

```
ValFilterErr(hval, APIFAMILY_DOS, API_DOSPOSTEVENTSEM, ERROR_ALREADY_POSTED,
             ERROR_ALREADY_POSTED);
```

See Also

ValInitialize (see page 55)

ValInitialize

HVAL ValInitialize*(pszAppName, pszLogFile, ulSupport)*

```
PSZ pszAppName;          /* Application Name          */
PSZ pszLogFile;          /* Log File Filename         */
ULONG ulSupport;         /* Validation Support Level  */
```

This function is used to register the application within the validation session and to request the level of support that the validation routines are to provide.

Parameters

⇓ **PSZ** *pszAppName*
Application filename or designation. The maximum size of the string that can be registered is 255 bytes. When requesting logging or ***ViewPort*** support, *pszAppName* cannot be NULL since the name or designation is used to distinguish applications that may be either logging to a common log file or tie into advanced filtering features of ***ViewPort***, otherwise the value can be NULL.

⇓ **PSZ** *pszLogFile*
Log file filename. The value is only used when you request log file support through the *ulSupport* parameter by setting support for either VL_ERRORLOG (0x01) or VL_VIEWPORTLOG (0x40). The maximum size of the filename that can be used is 255 bytes.

⇓ **ULONG** *ulSupport*
Validation support level requested. It can be any combination of the following values:

Symbol	Value	Meaning
VL_ERRORCODES	0x00	Provide validation return codes (default).
VL_ERRORLOG	0x01	Write errors in ASCII to error log specified in *pszLogFile* parameter. If this flag is used, the VL_VIEWPORTLOG (0x40) cannot be used since only one log file can be selected.
VL_FILELINE	0x02	Allow filename and line number to be included with each API call made. This is used in conjunction with VL_ERRORLOG (0x01). You must also make sure that INCL_VALAPI is defined to make

Symbol	Value	Meaning
VL_FILELINE (con't)		sure that the filename and linenumber information is recorded with each API. By not including the filename and linenumber in the ASCII log, the logging will be faster.
VL_PAUSELOG	0x04	Prevent errors from being immediately logged to the log file specified in the *pszLogFile* parameter. The **ValLogging** (see page 57) function can be used to turn on the logging at a later point.
VL_VIEWPORT	0x08	Send error information to ***ViewPort***.
VL_LOGRESET	0x10	Reset log file by deleting it. If you have allowed undeletion support on your machine, the file will be saved within the deleted files directory for the drive where the file is located.
VL_FORCELOGRESET	0x20	Reset log file by deleting it and preventing deleted file from being saved to the deleted files directory on the drive.
VL_VIEWPORTLOG	0x40	Write errors in ***ViewPort*** format to error log specified in *pszLogFile* parameter. If this flag is used, the VL_ERRORLOG (0x01) cannot be used since only one log file can be selected.

Return Value

A return value of zero (0) indicates are error return and validation will not occur to the level requested. Any other value is the handle for the validation session.

Comments

The routine is used to register the application formally with the validation DLL's that are being used. Depending on the flags specified within the *ulSupport* parameter, additional validation monitoring will be provided beyond the standard error return values which occurs even if the **ValInitialize** call is not used.

Example

To create a logging file that is to record the error information in ASCII, including filename and line numbers, you could use the following:

```
ValInitialize("Sample Application", "SAMPLE.LOG", VL_FILELINE | VL_ERRORLOG);
```

If you wanted to allow the information to be also sent to ***ViewPort***, you would change the above to:

```
ValInitialize("Sample Application", "SAMPLE.LOG", VL_FILELINE | VL_ERRORLOG |
              VL_VIEWPORT);
```

If you just wanted to have the error information sent to ***ViewPort***, you would use the following:

```
ValInitialize("Sample Application", NULL, VL_VIEWPORT);
```

See Also

ValLogging (see page 57)

ValLogging

ULONG ValLogging*(hval, fLogActivate)*

HVAL *hval*; /* Validation Handle */
BOOL *fLogActivate*; /* Logging Activate Flag */

This function is used to activate or deactivate logging.

Parameters

⇓ **HVAL** *hval*
Validation handle.

⇓ **BOOL** *fLogActivate*
When *fLogActivate* is a value of TRUE, any errors detected after the return of the **ValLogging** function will be recorded within the logging file specified in the **ValInitialize** function. If *fLogActivate* is a value of FALSE, any errors detected after the return of the **ValLogging** function will not be recorded within the logging file. Calling **ValLogging** a subsequent time with *fLogActivate* set to TRUE will reactivate the error logging to the log file.

Return Value

A return value of LOGERR_NONE (0) indicates error logging has been either activated or deactivated according to the value of *fLogActivate* otherwise it can be the following:

Symbol	Value	Problem
LOGERR_NO_LOG_REQUESTED	1	No logging file was defined through **ValInitialize**.
LOGERR_INVALID_HVAL	2	Invalid handle.

Comments The routine is used to activate or deactivate error logging to the file specified within the **ValInitialize** function. To determine if error logging is active, use the **ValQueryLogging** function.

Example To activate logging, you would use the following:

```
ValLogging(hval, TRUE);
```

To pause the logging, you would use the following:

```
ValLogging(hval, FALSE);
```

See Also **ValInitialize** (see page 55), **ValQueryLogging** (see page 59)

ValQueryClassMsgMonitor

ULONG ValQueryClassMsgMonitor*(hval)*

HVAL *hval*; /* Validation Handle */

This function is used to retrieve the class message monitor settings.

Parameters ⇓ **HVAL** *hval*
Validation handle.

Return Value A return value can be one of the following:

Symbol	Value	Meaning/Area
RCMMF_NONE	0x00000000UL	No message monitored.
RCMMF_FRAME	0x00000001UL	WM_* messages.
RCMMF_COMBOBOX	0x00000002UL	CBM_* messages.
RCMMF_BUTTON	0x00000004UL	BM_* messages.
RCMMF_MENU	0x00000008UL	MM_* messages.
RCMMF_STATIC	0x00000010UL	SM_* messages.
RCMMF_ENTRYFIELD	0x00000020UL	EM_* messages.
RCMMF_LISTBOX	0x00000040UL	LM_* messages.
RCMMF_SCROLLBAR	0x00000080UL	SBM_* messages.
RCMMF_TITLEBAR	0x00000100UL	TBM_* messages.
RCMMF_MLE	0x00000200UL	MLM_* messages.
RCMMF_SPINBUTTON	0x00004000UL	SPM_* messages.
RCMMF_CONTAINER	0x00008000UL	CM_* messages.
RCMMF_SLIDER	0x00010000UL	SLM_* messages.
RCMMF_VALUESET	0x00020000UL	VSM_* messages.
RCMMF_NOTEBOOK	0x00040000UL	BKM_* messages.
RCMMF_SKETCH	0x00100000UL	SKM_* messages.
RCMMF_GRAPHICBUTTON	0x00200000UL	GBM_* messages.
RCMMF_CIRCULARSLIDER	0x00400000UL	CSM_* messages.
RCMMF_ALL	0x007fffffUL	All messages.
REGMSG_INVALID_HVAL	0xffffffffUL	Invalid handle.

Comments The routine is used to retrieve the class message monitor selections that have been set through the **ValRegisterClassMsgMonitor** function. The

value returned will be a combination of the flags from the table above. The flags are OR'ed (|) together to complete the final value. You can check for a particular value by AND'ing (&) the symbol with the return value.

Example To determine the current class message monitor setting, you would use the following:

```
ulMsgClass = ValQueryClassMsgMonitor(hval);
```

See Also **ValInitialize** (see page 55), **ValRegisterClassMsgMonitor** (see page 59)

ValQueryLogging

ULONG ValQueryLogging*(hval)*

HVAL *hval*; /* Validation Handle */

This function is used to determine if logging is active or inactive.

Parameters

⇓ **HVAL** *hval*
Validation handle.

Return Value A return value can be one of the following:

Symbol	Value	Status
QLOG_NONE	0	Error logging was not requested through **ValInitialize**.
QLOG_ACTIVE	1	Error logging to logging file is active.
QLOG_INACTIVE	2	Error logging to logging file inactive.
QLOG_INVALID_HVAL	0xffffffffUL	Invalid handle.

Comments The routine is used to determine the current status of error logging. The function can be used in conjunction with **ValLogging** to activate or deactivate logging to the log file.

Example To determine logging status, you would use the following:

```
ulLogging = ValQueryLogging(hval);
```

See Also **ValInitialize** (see page 55), **ValLogging** (see page 57)

ValRegisterClassMsgMonitor

ULONG ValRegisterClassMsgMonitor*(hval, fl)*

HVAL *hval*; /* Validation Handle */
ULONG *fl*; /* Class Message Area Flag */

This function is used to set the class message monitor selections.

Parameters

⇓ **HVAL** *hval*
Validation handle.

⇓ **ULONG** *fl*
The area or areas that should have extended validation performed are defined through the *fl* flag. By combining the symbols below using the or (|) operator, you can describe more than one class message area. The value of *fl* used as the basis for checking the message parameters and returns for those areas. These areas are:

Symbol	Value	Area
RCMMF_NONE	0x00000000UL	No message monitored.
RCMMF_FRAME	0x00000001UL	WM_* messages.
RCMMF_COMBOBOX	0x00000002UL	CBM_* messages.
RCMMF_BUTTON	0x00000004UL	BM_* messages.
RCMMF_MENU	0x00000008UL	MM_* messages.
RCMMF_STATIC	0x00000010UL	SM_* messages.
RCMMF_ENTRYFIELD	0x00000020UL	EM_* messages.
RCMMF_LISTBOX	0x00000040UL	LM_* messages.
RCMMF_SCROLLBAR	0x00000080UL	SBM_* messages.
RCMMF_TITLEBAR	0x00000100UL	TBM_* messages.
RCMMF_MLE	0x00000200UL	MLM_* messages.
RCMMF_SPINBUTTON	0x00004000UL	SPM_* messages.
RCMMF_CONTAINER	0x00008000UL	CM_* messages.
RCMMF_SLIDER	0x00010000UL	SLM_* messages.
RCMMF_VALUESET	0x00020000UL	VSM_* messages.
RCMMF_NOTEBOOK	0x00040000UL	BKM_* messages.
RCMMF_SKETCH	0x00100000UL	SKM_* messages.
RCMMF_GRAPHICBUTTON	0x00200000UL	GBM_* messages.
RCMMF_CIRCULARSLIDER	0x00400000UL	CSM_* messages.
RCMMF_ALL	0x007fffffUL	All messages.

Return Value

A return value can be one of the following:

Symbol	Value	Meaning/Area
REGMSG_NONE	0UL	Normal return
REGMSG_INVALID_FLAG	1UL	Invalid flag value.
REGMSG_INVALID_HVAL	0xffffffffUL	Invalid handle.

Comments

The routine is used to set the class message monitor selections that will have the message parameters validated along with the message return for **WinSendDlgItemMsg** and **WinSendMsg** API's. By default, no monitoring is performed since the monitor of the various messages can have significant impact on performance. The value of the parameter *fl*

is not cumulative. Therefore, the value that you provide is used to replace the previous value. The current value can be determined by **ValQueryClassMsgMonitor**.

Example

To allow the monitoring of the messages sent to various controls, like an entry field and list box, you would use the following:

```
ValRegisterClassMsgMonitor(hval, RCMMF_ENTRYFIELD | RCMMF_LISTBOX);
```

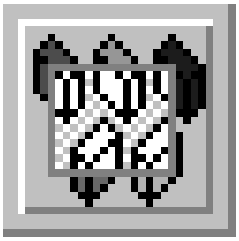
You can, if you so desire only provide the monitoring of messages in specific areas of your code. To allow this to happen, you could use the above example and when you want to disable the monitoring, you would use the following:

```
ValRegisterClassMsgMonitor(hval, RCMMF_NONE);
```

See Also

ValInitialize (see page 55), **ValQueryClassMsgMonitor** (see page 58)

Using ViewPort



Introduction

This section describes how you use ***ViewPort*** to provide real-time monitoring of errors returned through the validation routines and the actual API's themselves. To allow the maximum potential of ***ViewPort*** to help you during the development process, you need to compile your source code such that the maximum validation information can be provided to the validation DLL's.

To do this, you need to define the symbol INCL_VALAPI before the OS2.H file is included in your source files. You can achieve this by doing the following in your code:

Figure 4.1
INCL_VALAPI definition

```
#define INCL_DOS                    /* Include OS/2 DOS Kernel                */
#define INCL_WIN                    /* Include OS/2 PM Windows Interface      */

#define INCL_VALAPI                 /* Include Validator Support              */

#include <os2.h>

#include "appdefs.h"
#include "threads.h"
```

Alternately, an easier method of defining the symbol is to include /DINCL_VALAPI with your compiler switches. This will define the symbol and you will not have to edit your source code to define the symbol.

By defining the INCL_VALAPI symbol before the OS2.H file is included in your source module, you will enable OS/2 API's to be redefined such that they include two extra parameters in each call. These two parameters will always be the last two parameters. The best way understand this need is to see what happens with a typical OS/2 API.

The OS/2 API **DosCreateDir** is defined as:

Figure 4.1 **DosCreateDir** API

```
APIRET DosCreateDir(   PSZ pszDirName, PEAOP2 peaop2    );
```

The redefinition components of the validation headers are conditionally compiled and when the INCL_VALAPI symbol is defined, the **DosCreateDir** is redefined to:

Figure 4.2 **DosCreateDir** API redefinition

```
APIRET _DosCreateDir( PSZ pszDirName, PEAOP2 peaop2, PSZ pszFilename,
                      ULONG ulLine  );
```

You don't have to worry about changing you source code to include the last two parameters since the headers provided with ***Validator*** will do this for you. The best way to see this is with the actual usage of the call:

Figure 4.3 **DosCreateDir** API usage example

```
DosCreateDir("NewDir", (PEAOP2)NULL);
```

Ultimately, when the actual **DosCreateDir** API is encountered within your source code, the final result as produced by the preprocessor would be:

Figure 4.4 **DosCreateDir** API redefinition preprocessed result

```
_DosCreateDir("NewDir", (PEAOP2)((void *)0), "Test.C", 11);
```

Now that you understand what happens to your source code when it is compiled with the INCL_VALAPI defined, it is only a matter of making the connection between this and ***ViewPort***.

When an error is detected through the validation routines either as a parameter error or as returned through the normal processing of the API call, the validation routines prepare a package of information that is then transferred to ***ViewPort*** for processing. Two items within this package are the name of the file where the API was called from and the actual line number. This information is essentially the last two parameters as defined above. And, when you see the error information recorded within an error monitoring window of ***ViewPort***, you can immediately see which file and the line in that file where the error occurred.

But, this is not the only thing that you need to do within the source code to enable ***ViewPort*** to be used. You also need to issue a call to the validation routines essentially initializing the routines for the ***ViewPort*** communication. This call, **ValInitialize** (see page 55), through the *ulSupport* parameter allows you to request ***ViewPort*** support by including the flag VL_VIEWPORT (0x08).

This causes the validation DLL's to start the communication process with ***ViewPort***, thereby informing ***ViewPort*** that an application is

utilizing validation support and wants it to be monitored real-time. This implies that ***ViewPort*** must being running for this to be successful. If you have requested ***ViewPort*** support through the **ValInitialize** function and ***ViewPort*** is not running, the following message will be displayed:

Plate 4.1
ViewPort not running message

As the message states, by starting ***ViewPort*** and then clicking the mouse pointer on the OK pushbutton, the communication link between the validation DLL's and ***ViewPort*** can still occur.

Once the communication link is made, ***ViewPort*** opens an error window unless you have asked for appending within existing error windows that have the same application name. ***ViewPort*** then waits for packets from the validation DLL's containing error information and when it receives it, ***ViewPort*** then adds it to the list of errors recorded within the window and then displays the information at the bottom of the list.

As described, you can see that ***ViewPort*** needs a little help from you for it to successfully operate. Even if you don't provide the INCL_VALAPI definition but instead use the INCL_VAL definition you can still request ***ViewPort*** support through the **ValInitialize** function. Error information will still be packaged and sent to ***ViewPort*** except that it won't have the filename and line number from which the OS/2 API was called. This may make it a little more difficult to track down the problem, but at least you know that you have specific problem with a given API.

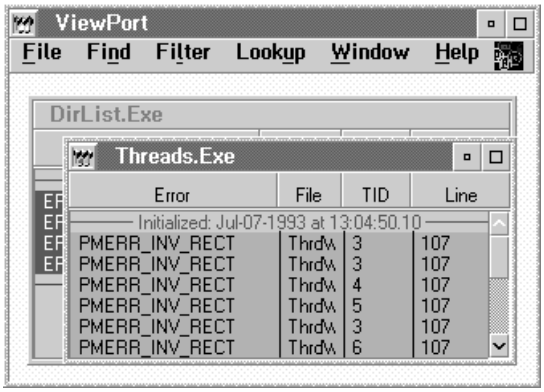
ViewPort is not an overly complicated tool from your perspective in that you need to learn complex combinations of menus and accelerator keys to make it operate successfully. All you need to do for the most part is start it and wait.

This waiting does not necessarily have to be with only one of your applications you are developing, it can be with many. As long as you have used the **ValInitialize** with distinct application names, ***ViewPort***

can monitor as many different applications you have created with validation support. ***ViewPort***, depending on how you have configured it, will open a separate error monitoring window for each different application you register through the **ValInitialize** call. When an error is detected within any of the applications, the error information is sent from the validation routines to the correct error monitoring window, where again depending on how you have configured ***ViewPort***, will immediately display the error information within the window.

Plate 4.2 shows ***ViewPort*** with two error monitoring windows active and with each have errors recorded within them.

Plate 4.2
ViewPort window



The title bars of each error window will contain the name you used as the application name within the **ValInitialize** call. Figure 4.3 shows an individual error monitor window. You will notice that it is divided into four columns. The first column contains the error symbol for the error detected. Even if the error detected by the validation routines is a parameter error, ***ViewPort*** will always show the returned system error component. The method that is used to distinguish between parameter errors and API return errors is through colour. The colours by default are:

Table 4.1
Colour options

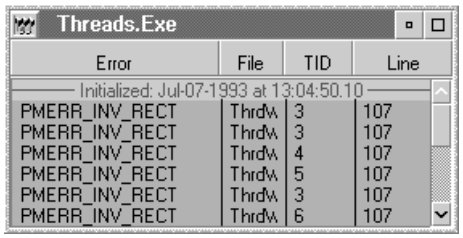
Option	Default colour
Parameter error text	Black
Parameter error background	Cyan

Option	Default colour
Return error text	White
Return error background	Dark cyan
Invalid pointer text	Black
Invalid pointer background	Pale gray
Filtered error text	Black
Filtered error background	Yellow

They can be redefined through the **ViewPort Configure** dialogue (see page 75) which not only allows you to define the colours used, but also allows you to define whether or not applications using the same application name are able to append to a window that is already open or whether a new window must be opened, and to allow for certain types of errors to have alarms.

The second column is used for the name of the file that used the API call. If you did not compile the source code with the INCL_VALAPI defined before the inclusion of the OS2.H file, this will be blank. It will also be blank if a C or C++ runtime library routine called an OS/2 API that either caused a parameter error or API return error.

Plate 4.3
Threads.Exe error window



The third column is used to denote the thread ID from which the API call was made. It should be explained here how you interpret this column. Thread 1 is always the first thread in the application. It is associated with the main( ) function and it is from this thread that secondary threads within the application will be started.

If you issue the **ValInitialize** call immediately upon entry into the main( ) function of your application, a second thread will be started by the **ValInitialize** call for the use of the validation routines. This is done

specifically to allow the communication between the validation routines and ***ViewPort*** such that it does not tie up your main thread and any secondary threads that you start. This thread would always be thread 2.

Any secondary threads that you would start within your application would begin from thread 3 and increment upwards. Therefore, if you have two threads within your application and you see thread 1 and thread 3 with errors, thread 2 is the thread used by the validation routines and everything is within reason.

As you can see in Plate 4.2, the threads start at 3 and increment upwards from there.

The only time this would be different is when you issue a **ValInitialize** after you have started other threads within your application.

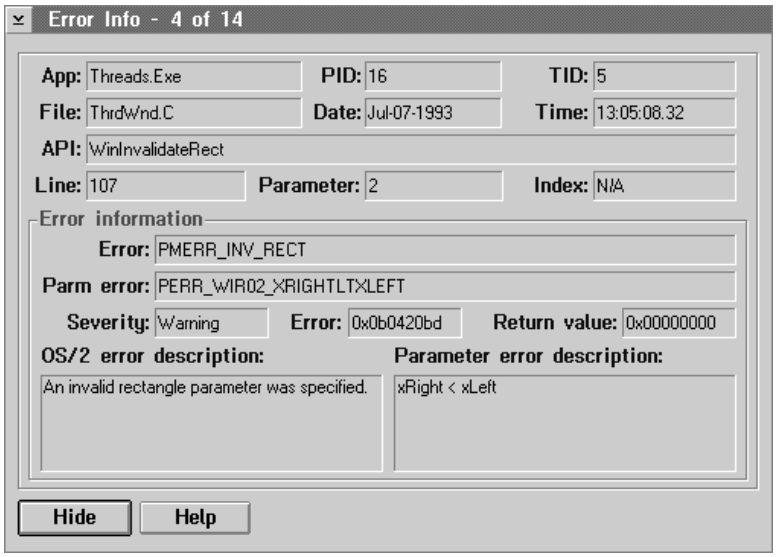
Therefore, if it appears as though the numbering of the threads is off by one, it is only when you are using the validation DLL's with your application.

The final column contains the line number from which the API call was made in your source code. Again, like the filename column, if you did not compile the source code with the INCL_VALAPI defined before the inclusion of the OS2.H file, this will be blank. It will also be blank if a C or C++ runtime library routine called an OS/2 API that either caused a parameter error or API return error.

Error Info window

Although you think that the error monitor window supplies more than enough information to allow you to correct problems within your source code if they do exist, ***ViewPort*** has even more information for the entry within the list. To view this information all you need to do is double click the mouse pointer on the line in question within the error monitoring window. This will cause the **Error Info** window to be displayed, Plate 4.3.

Plate 4.4
Error Info window



The information contained within this window is divided into two areas. The first area is contained within the upper half of the window. It contains the following pieces of information:

Table 4.2
Error Info window general items

Area	Contents
App	Contains the application name as registered through the **ValInitialize** call.
PID	Contains the process ID of the application.
TID	Contains the thread ID from which the API call was made.
File	Contains the name of the source code module from which the call was made. This will only be displayed if the source code was properly compiled as described above. By defining the symbol INCL_VALAPI before the inclusion of the OS2.H file will enable this to occur.
Date	Contains the date the error was detected on. The purpose of this is when you have saved the error information to a log file or have received a log file from a user you can tell when the error occurred.

Area	Contents
Time	Contains the time the error was detected at. The purpose of this is when you have saved the error information to a log file or have received a log file from a user you can tell when the error occurred.
API	Contains the proper name of the API that the error was received for.
Line	Contains the line within the source code module from which the call was made. This will only be displayed if the source code was properly compiled as described above. By defining the symbol INCL_VALAPI before the inclusion of the OS2.H file will enable this to occur.
Parameter	Contains the index of the parameter that was detected to be in error by the validation routines. When the error was returned by the actual API call, this will contain N/A. The numbering of the parameters is the first parameter from the left is number 1, the second parameter from the left is number 2, and so on. Also, if you are using the IBM C Set/2 or C Set++ compilers, additional parameter checking is performed (***this can only be done with the IBM compiler*** since it is the only compiler that fully conforms with the system linkage calling conventions as described on page 102). When too few parameters are detected for the API call, it will denote the number of parameters that are missing. When too many parameters are detected, it will denote the number extra that were found.
Index	Contains the array index that was detected to be in error for the parameter. For example, the third parameter of the **GpiPolyline** call an array of points (array of **POINTL** data types). If the index value was 3 and you had defined the array as POINTL aptl[5]; the error would be for aptl[3].

The second part of the window contains the detailed error information. It contains the following pieces of information:

Table 4.3
Error Info window error items

Area	Contents
Error	Contains the OS/2 defined error symbol for the error returned.
Parm error	Contains the parameter error symbol that is defined within the ValErrs.H file. This field will only contain information when a parameter error is detected through the validation routines.
Severity	Contains the severity for the error. This will only contain a value for Ddf*, Dev*, Drg*, Gpi*, Pic*, Prf*, some Spl* and Win* calls. The levels that will be shown are: **Warning**, **Error**, **Severe**, and **Unrecoverable**.
Error	Contains the numeric error value for the error being reported. The format of the error value will conform to that as described on page 10 for the API type.
Return value	Contains the return value for the API call. In the case of Dos*, Prt*, some Spl*, Pen for OS/2 and MMPM/2 calls, the error value will be the same as the return value.
OS/2 error description	Contains the OS/2 error description for the error The purpose of this is when you have saved the error information to a log file or have received a log file from a user.
Parameter error description	Contains the parameter error description for the error only if a parameter error was detected.

You can leave the window displayed and select another item within the error list or for that matter, select a different error list and an item from within it.

You will find that for the most part the information contained within the error monitoring windows will be more than sufficient for helping you to solve your problems. The **Error Info** window will most likely only be used when you can't see what the problem is with the API call as you have used it within your source code.

Even though the information provided is in as much detail as possible, you will find that you will still have to utilize the services of the debugger since some of the more difficult errors to track involve values

returned back to you by other OS/2 API's or your own functions or calculations you make in the code preceding the call.
Having finished this short tour of the error monitoring windows, now begins the tour around the actual ***ViewPort*** menu which you can use to augment the usage of the above.

ViewPort action bar

Through the action bar pull down menu's, you can perform commands and view information. The action bar menus provided are:

Plate 4.5
ViewPort action bar

Table 4.4
Action bar menu items

Menu	Contents
File	Contains the file oriented commands such as Open, Close, Save As, Print and Configure. These commands allow you open and save log or filter files along with the printing of them.
Find	Contains the commands that allow you to search for a specific error or API within one of the open error windows.
Filter	Contains the commands that allow you to enable or disable error filtering and to select the filter files that are to be preloaded when ***ViewPort*** starts.
Lookup	Contains the commands that allow you to lookup the error descriptions for either an error value or symbol.
Window	Allows you to manage open error windows within the ***ViewPort*** window.
Help	Contains the commands to invoke the help information for ***ViewPort***.

File menu

The **File** menu is used in conjunction with file operations and the **File** sub-menus provided are:

Plate 4.6
File menu

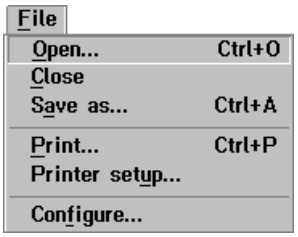


Table 4.5 File menu items

Menu Item	Purpose/Usage
Open	Used to open an existing error log or filter file (see page 79).
Close	Used to close the current error window.
Save as	Used to name and save an error log file or filter file to disk as well as edit history, comments and version information associated with the file (see page 80).
Print	Used to print the current error monitoring window (see page 82).
Printer setup	Used to select printer and/or setup printer (see page 84).
Configure	Used to configure ***ViewPort*** for the colours, alarms and window handling methods (see page 75).

Find menu

The **Find** menu is used in conjunction with find API and error operations and the **Find** sub-menus provided are:

Plate 4.7 Find menu

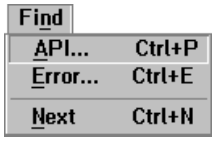


Table 4.6 Find menu items

Menu Item	Purpose/Usage
API	Used to search for a selected API (see page 85).
Error	Used to for a selected error (see page 87).
Next	Used to search for either the last API or error depending on the last search operation performed.

Filter menu

The **Filter** menu is used in conjunction with error filtering and the **Filter** sub-menus provided are:

Plate 4.8 Filter menu

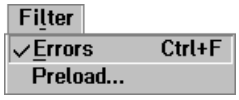


Table 4.7
Filter menu items

Menu Item	Purpose/Usage
Errors	Used to enable or disable error filter for the error window current displayed.
Preload	Used to select the filter files that are to be preloaded when ***ViewPort*** starts (see page 89).

Lookup menu

The **Lookup** menu is used to lookup error descriptions for either an error value or symbol and the **Lookup** sub-menus provided are:

Plate 4.9
Lookup menu

Table 4.8
Lookup menu items

Menu Item	Purpose/Usage
Error	Used to lookup the description of an error value return by an API call (see page 91).
Error description	Used to lookup the description of an OS/2 error symbol (see page 93).

Window menu

The **Window** menu is used in conjunction with monitoring window selections and arrangements. The **Window** sub-menus provided are:

Plate 4.10
Window menu

Table 4.9
Window menu items

Menu Item	Purpose/Usage
Close all	Closes all open error windows..
Tiled	Tiles source error windows.
Cascaded	Cascades source error windows.

Help menu

The **Help** menu provides you with access to on-line help for ***ViewPort***. The menu items available are:

Plate 4.11
Help menu

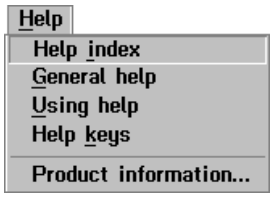


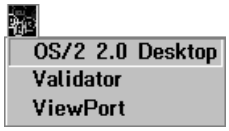
Table 4.10
Help menu items

Menu Item	Purpose/Usage
Help index	Used to display the help index.
General help	Used to display general help for the application.
Using help	Used to display help for using the help.
Keys help	Used to display help for keys.
Product information	Used to display information dialogue on ***ViewPort***.

Window-list menu

The **Window-list** menu allows you to select other applications that are currently running along with any folders that you may have currently open. The menu items could appear like:

Plate 4.12
Window list menu



Configuring ViewPort

Even though ***ViewPort*** is easy to use when you first run it, you still may want to tailor some of the aspects of how it operates. For example, you may want to use colours within the error monitoring windows that have significant meaning to you instead of the defaults provided. You can also set alarms for the major error types such that when an error occurs of a specific type, you will receive an audible notification along with the entry within the error monitoring window.

The **ViewPort Configure** dialogue, which allows you to configure ***ViewPort***, consists of a notebook with three pages of options. The first page, **Options**, allows you to set the base options. The second page, **Colours**, allows you to set the display colours of errors within the error monitoring windows. The final page, **Alarms**, allows you to set the error types in which you want to have audible indication they have occurred.

The first page, **Options**, provides the following options which affect the operation of the error monitoring window. It is divided into two parts:

Error window and **App registration**. The **Error window** component provides the following option:

Table 4.11
Error window options

Option	Purpose
Scroll window	Used to allow the error monitoring windows to automatically scroll the contents up when a new error entry is added such that the entry is immediately view able. The default is for the window not to scroll.

Plate 4.13
ViewPort Configure dialogue showing Options notebook page

The **App registration** group provides the following options:

Table 4.12
App registration options

Option	Purpose
Append	Used to cause ***ViewPort*** to search open error monitoring windows for a matching application name when a new application is registered through the **ValInitialize** function (see page 55). The first parameter of **ValInitialize**, *pszAppName*, is used as the matching criteria and is case sensitive. If a match occurs, any errors from the newly registered application will be placed within the matched window, otherwise a new window will be opened and the errors will be placed within that window.

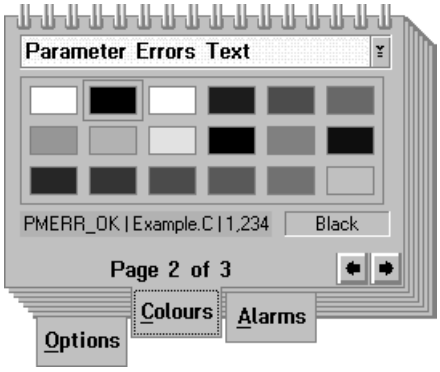
Option	Default colour
New window	Used to cause ***ViewPort*** to open a new window for each application registered through the **ValInitialize** function. This is the default.

The **Colours** notebook page allows you to set the colours for the following:

Table 4.13
Colour options

Option	Default colour
Parameter error text	Black
Parameter error background	Cyan
Return error text	White
Return error background	Dark cyan
Invalid pointer text	Black
Invalid pointer background	Pale gray
Filtered error text	Black
Filtered error background	Yellow

Plate 4.14
ViewPort Configure dialogue - Colours notebook page



You select the area to set the colour for from the drop-down list above the colour grid. Below the colour grid is a sample of what the result would appear as in the error monitoring window. The name of the colour selected will appear beside the sample.

Changing the colour for the selected area is accomplished by clicking the mouse pointer on the square with the desired colour. When you do select

the desired colour, the sample area will reflect the change and the name of the colour will be shown beside the sample.

The final page within the notebook, **Alarms**, allows you to set audible indicators that are sounded when that particular error type is received from the validation routines. The alarms that can be set are:

Plate 4.15
ViewPort Configure dialogue - Alarms notebook page

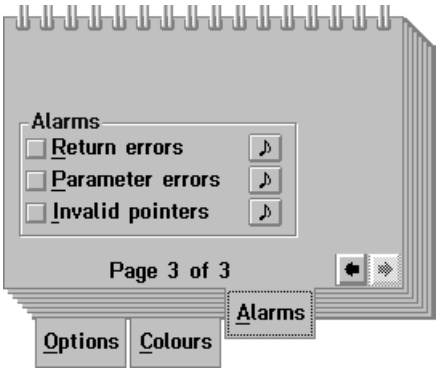


Table 4.14
Alarms

Option	Alarm Type
Return errors	Sounds an alarm when a return error is received.
Parameter errors	Sounds an alarm when a parameter error is received.
Invalid pointers	Sounds an alarm when an invalid pointer is passed as a parameter to an OS/2 API.

All alarms are off by default.

The sound that each alarm will make can be heard by clicking the mouse pointer on the push button beside each option. As you will notice, each alarm provides a distinct two-tone sound.

The most useful alarm is mostly likely going to be for the invalid pointer. Not only does it quickly alert you, it will allow you to take remedial action in case the failure of the OS/2 API causes your application to become unstable.

N O T E

Even though you can set alarms for both return and parameter errors, you may want to limit the usage of each depending on the number of errors you may expect as your speaker may continually be active and eventually may cause you and others around you some irritation.

File support

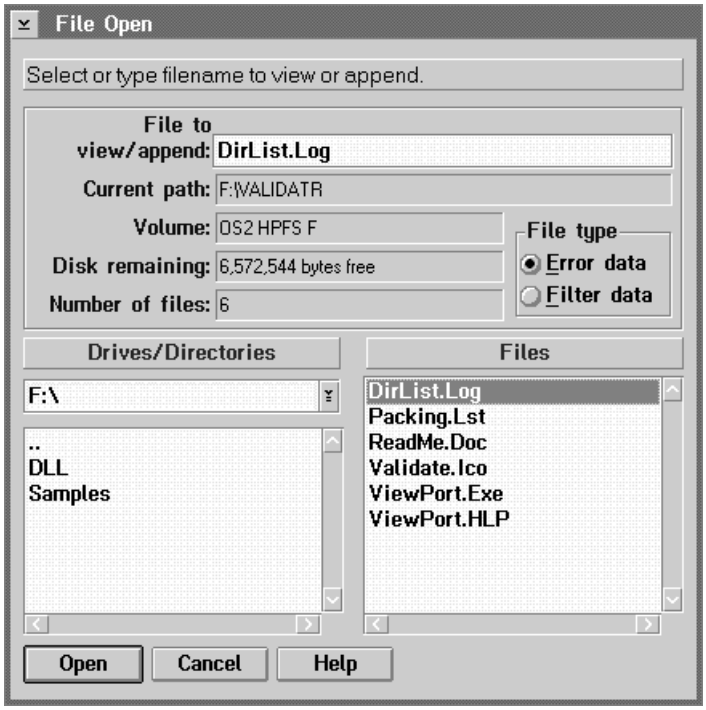
Validator allows you to save the error information to a logging file when you request logging support through the **ValInitialize** function. ***ViewPort*** can read one of the logging file types as though the information had been originally received by it. Also, ***ViewPort*** allows you to save the information from an error monitoring window such that it can be opened and viewed at a later point like that of a logging file. It also allows you to select errors to filter out (see page 89) and save the filtered errors to a file that can be used to automatically filter errors you are not particularly interested in for that particular application.

Along with the ability to save and retrieve error information, ***ViewPort*** allows you print the information in a variety of formats. It will allow you to select the output device along with the fonts and sizes.

Opening error logs and filter files

One of the file oriented dialogues is the **File Open** dialogue. From this dialogue you can select the drive and directory where the error or filter file is located, and to select the error or filter file to load.

Plate 4.16
File Open dialogue



You select a drive or directory to change to by double clicking the mouse pointer on the entry within the **Drives/Directories** list box within the dialogue. Upon changing to that drive or directory, a list of files

contained within that directory will be displayed within the **Files** list box. You can select the file to open by simply clicking on the entry within the **Files** list box, typing in the name within the entry field labeled **File to view/append** or by double clicking the mouse pointer on the entry in the list box. This later operation is equivalent to clicking the mouse pointer on the entry and then on the **Open** push button.

You can open one of two file types:

Table 4.15 File types

Option	File Type
Error data	File contains error data that was created either through the validation routines logging facilities in ***ViewPort*** data format or from a ***ViewPort*** error monitoring window. The plain ASCII logging files cannot be opened by this function.
Filter data	File contains error filter data which is essentially a list of errors that you do not want an error monitoring window to display when it is received from the validation routines when you have allowed error filtering to occur (see page 89).

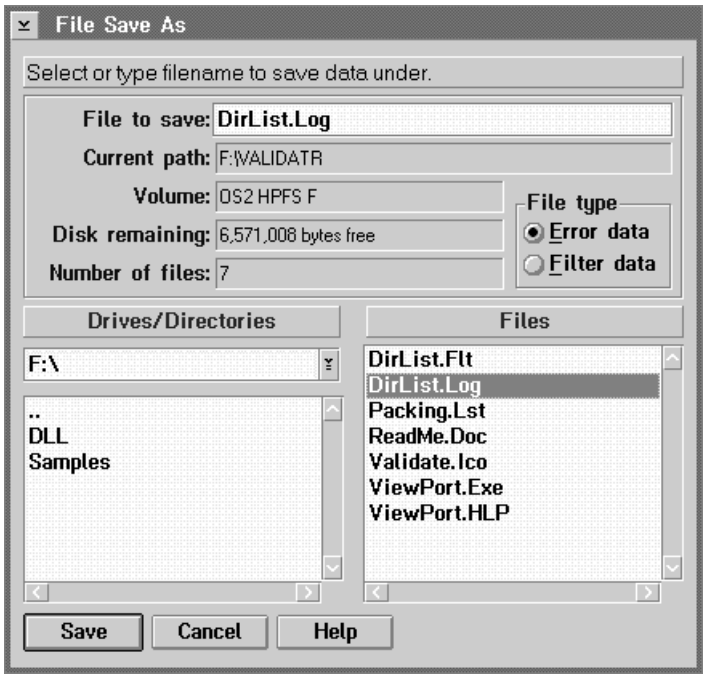
Before you either double click the mouse pointer on the file you wish to open within the **Files** list box or click on the **Open** push button, you should make sure that you have selected the file type. If you select the wrong type, ***ViewPort*** will automatically read in the file for the correct type recorded within the file instead of that you may have requested.

Saving error logs and filter data

When you want to save the information contained with an error monitoring window, you can use the **File Save As** dialogue. It is similar in appearance to the **File Open** dialogue except here you save information to the disk instead of retrieve information from the disk.

You select a drive or directory to change to by double clicking the mouse pointer on the entry within the **Drives/Directories** list box within the dialogue. Upon changing to that drive or directory, a list of files contained within that directory will be displayed. You can select the filename to save to by simply clicking on the entry within the **Files** list box, typing in the name within the entry field labeled **File to view/append** or by double clicking the mouse pointer on the entry in the list box. This later operation is equivalent to clicking the mouse pointer on the entry and then on the **Save** push button.

Plate 4.17
File Save As dialogue



You can save one of two file types:

Table 4.16
File types

Option	File Type
Error data	File contains error data.
Filter data	File contains error filter data which is essentially a list of errors that you do not want an error monitoring window to display when it is received from the validation routines when you have allowed error filtering to occur (**Filter Errors** menu item).

Before you either double click the mouse pointer on the file you wish to save under within the **Files** list box or click on the **Save** push button, you should make sure that you have selected the file type.

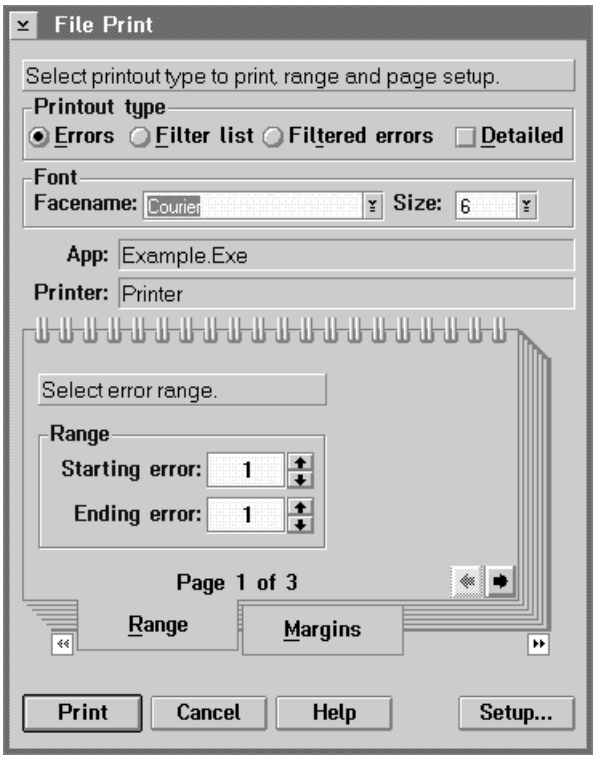
Unlike the **File Open** dialogue, you must ensure you have selected the correct data type to save under since the information saved records the file type within it. If you select the wrong type, the information will be saved under the wrong format.

Printing error and filter information

If you need hard copy of the information within an error monitoring window, you can use the **File Print** dialogue to print out the type of information that you may require.

Through the dialogue you can set the font face name and size to print under along with the error entry range, margins, header and footer titles.

Plate 4.18
File Print dialogue showing Range notebook page



You select the type of printout from options at the top of the dialogue under the group labeled **Printout type**. These options are:

Table 4.17
Printout types

Option	Printout Type
Errors	Printout will contain all errors received within the error monitoring window.
Filter list	Printout will contain a list of errors that are being filtered for within the error monitoring window.
Filtered errors	Printout will contain only errors have been filtered and are visible within the error monitoring window.

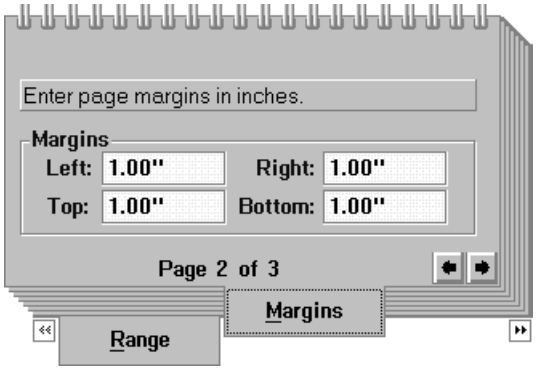
Option	Printout Type
Detailed	This is used in combination with the above options in that when this option is not selected, the information printed will be similar to the format contained within the error monitoring windows where only the error, filename, thread ID and line number will be printed. When the option is selected, detailed error information like that contained within the **Error Info** window (see page 68) will be printed.

You select the font and font size to use through the drop-down lists that are located within the **Font group**. The drop-down labeled **Facename** will contain a list of all the fonts supported by the printer you have selected. The current font selected will be displayed and the corresponding sizes available for that face name will appear within the drop-down list labeled **Size**.

Below the **Font** group are two fields which will show the error monitoring window application name and the printer that will be used for printing the information on. In the case of the printer, if you wish to change the printer selection, you can use the **Setup...** push button in the bottom right corner of the dialogue to select a new printer or, change the printer options through the **Printer Setup** dialogue (see page 84).

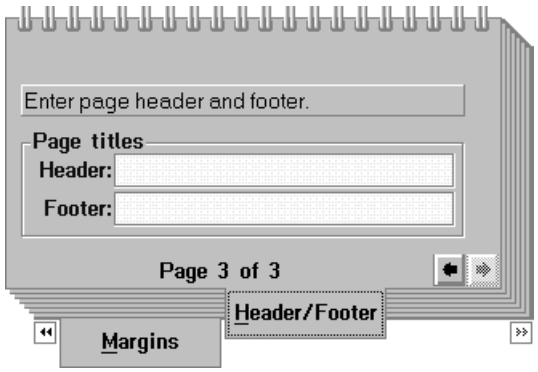
Options for the error range, margins and header/footer titles are selected through three pages contained within a notebook. The first page of the notebook, **Range**, allows you to define the range of error lines that you wish to print. Through the two spin buttons, you will be able to select the line range.

Plate 4.19
File Print dialogue - Margins notebook page



The second notebook page, **Margins**, allows you to set up the left, right, top and bottom margins which the printout must be bounded to. The measurement for the margins is in inches and you can enter fractions if desired. For example, if you want the left margin to be one-half inch, you would enter 0.5 in the entry field labeled **Left**.

Plate 4.20
File Print dialogue - Header/Footer notebook page



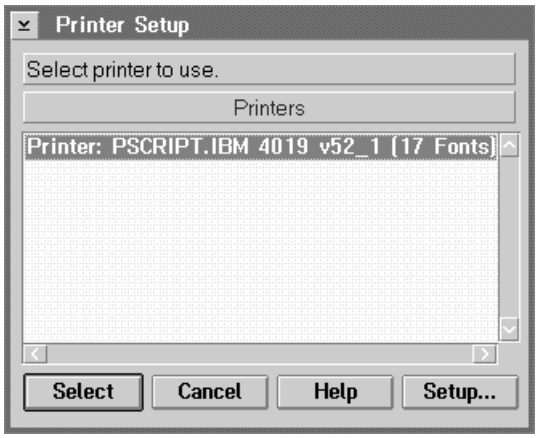
The last notebook page, **Header/Footer**, is used to enter the header and footer titles that will appear on each page printed. The text entered within the entry field labeled **Header** will be printed within the top margin of the page and the text entered within the entry field labeled **Footer** will be printed within the bottom margin of the page. In both cases, the maximum number of characters that can be entered is 255. If the number of characters entered is greater than can be printed, as much of the title that can be printed as possible within the header or footer area will be printed.

Printer setup

The **Printer Setup** dialogue not only allows you to select the device in which to send the printed output of the file selected to, but also allows you to setup the device.

You select the printer or output device to use by simply selecting the device from the list box. You can further set up the device by selecting the **Setup...** push button after which device specific dialogues will be displayed thereby allowing you to configure the device or select device options.

Plate 4.21
Printer Setup dialogue



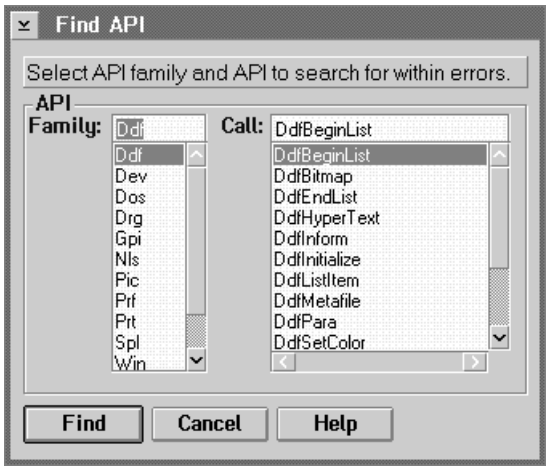
Finding errors or API's

ViewPort provides facilities to allow you to search an error monitoring window for a specific error or API. This can be useful when you have an error monitoring window that has hundreds or even thousands of entries.

Finding an API

To find an OS/2 API, you use the **Find API** dialogue. You pick the OS/2 API by family and API name. The family is selected from the drop-down list labeled **Family**.

Plate 4.22
Find API dialogue



The drop-down will contain the following family types:

Table 4.18
API family types

Designation	Family
Ddf	All calls prefixed by **Ddf**. These calls belong to the Dynamic Data Formatting API's.

Designation	Family
Dev	All calls prefixed by **Dev**. These calls belong to the Device API's.
Drg	All calls prefixed by **Drg**. These calls belong to the Drag and Drop API's.
Dos	All calls prefixed by **Dos**. These calls belong to the kernel API's.
Gpi	All calls prefixed by **Gpi**. These calls belong to the Graphics Programming Interface API's.
Nls	All calls prefixed by **Nls**. These calls belong to the Nation Language Support API's. These calls at the present time cannot be monitored through ***Validator*** since they are still 16-bit.
Pic	All calls prefixed by **Pic**. These calls belong to the Picture Interchange API's.
Prf	All calls prefixed by **Prf**. These calls belong to the Profile API's.
Prt	All calls prefixed by **Prt**. These calls belong to the Print API's.
Spl	All calls prefixed by **Spl**. These calls belong to the Spooler API's.
Win	All calls prefixed by **Win**. These calls belong to the Window API's.
MMPM/2	All calls prefixed by **Spi**, **mci** and **mmio**. These calls belong to the MMPM/2 API's.
Pen	All calls prefixed by **Red**, **Vkp** and **Wrt**. These calls belong to the Pen for OS/2 API's.

When you select an entry from the **Family** drop-down, the drop-down labeled **Call** will be cleared of the current entries and replaced with the valid API's for the family selected.

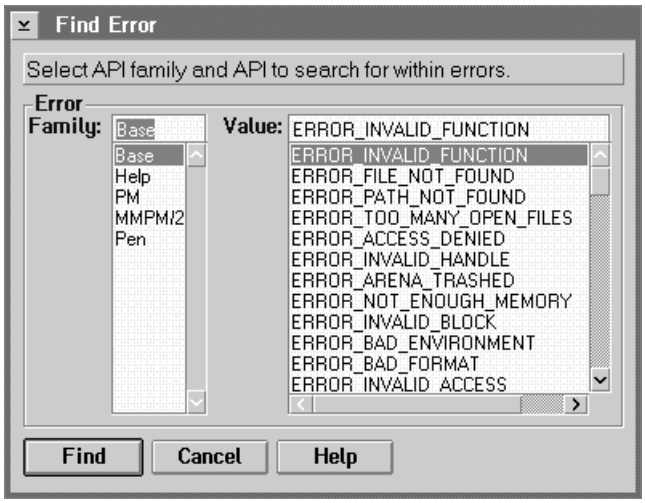
You then select the API to search from that list. Clicking the mouse pointer on the **Find** button will cause ***ViewPort*** to search the current window for the API and if it is found within the error monitoring window, the first occurrence will be selected and displayed within the window. If the API cannot be found, a message will be displayed informing you.

An alternate method of selecting the API and then clicking the mouse pointer on the **Find** button is to double click the mouse pointer on the entry within the **Call** drop-down.

Finding errors

To find an error, you use the **Find Error** dialogue. You pick the OS/2 error by family and value name. The family is selected from the drop-down list labeled **Family**.

Plate 4.23
Find Error dialogue



The drop-down will contain the following family types:

Table 4.19
Error family types

Designation	Family
Base	All errors prefixed by ERROR_. These errors belong to the Dos*, Prt* and some Spl* API's.
Help	All errors prefixed by HMERR_. These errors are the result of the following API's: **WinAssociateHelpInstance**, **WinCreateHelpInstance**, **WinCreateHelpTable**, **WinDestroyHelpInstance**, **WinLoadHelpTable**, and **WinQueryHelpInstance**.
PM	All errors prefixed by PMERR_. These errors belong to the Ddf*, Dev*, Drg*, Gpi*, Pic*, Prf*, some Spl* and most Win* except those noted in the Help family API's.

Designation	Family
MMPM/2	All errors prefixed by ERROR_, MCIERR_ AND MMIOERR_. These errors belong to the MultiMedia for PM/2 (MMPM/2) API's.
Pen	All errors prefixed by REDERR_, VKPERR_ and WRTERR_. These errors belong to the Pen for OS/2 API's.

When you select an entry from the **Family** drop-down, the drop-down labeled **Value** will be cleared of the current entries and replaced with the valid error designations for the family selected.

You then select the error designation to search from that list. Clicking the mouse pointer on the **Find** button will cause ***ViewPort*** to search the current window for the error designation and if it is found within the error monitoring window, the first occurrence will be selected and displayed within the window. If the error designation cannot be found, a message will be displayed informing you.

An alternate method of selecting the error designation and then clicking the mouse pointer on the **Find** button is to double click the mouse pointer on the entry within the **Value** drop-down.

When searching for either an error or API, the search starts from the current selection within the error monitoring window. Searching for the next occurrence of the error or API can be done by selecting the **Find Next** menu item or pressing Ctrl+N on the keyboard. The search again is started from the current selection.

Filtering errors

ViewPort allows you to filter the input from the validation DLL's such that only those errors that you may not be expecting are displayed. This filtering process is not used in place of the filtering within the validation DLL's where filtering there is performed to prevent overloading the messaging between the validation DLL's and ***ViewPort***.

The main purpose of filtering and usage is to select errors within a error message window that you know you will always receive and are part of your applications design. An example of this is the **DosFindFirst**, **DosFindNext** API combination. When the search of the current directory is complete, **DosFindNext** will return an error, ERROR_NO_MORE_FILES, which your application would then interpret as the end of the directory search. Your application would most likely fall out of a loop and continue on.

Since you expect this error, you may want ***ViewPort*** to prevent this error from being displayed within the error monitoring window when it is received. To allow this, you must perform the following actions.

First, you must allow the application you are monitoring to run at least one time, having the application execute the API that issues the error in the line and module where that error is expected. The error information will be received by ***ViewPort*** in the normal manner and processed such that it is displayed within the error monitoring window. When you have finished running the application, you then locate the error within the window and while pressing the Ctrl key on the keyboard, you also press button 1 of the mouse while the mouse pointer is overtop the entry in the list. It will then change to the filtered error colour allowing you to see that you have selected the error entry for filtering purposes.

Once you have selected the errors you wish to filter for, you should then select the **File Save As** dialogue (see page 80) to save the filter information to your hard disk or network. Make sure that when you save the information you select the **Filter data** radio button to save the filter information.

Now, if you select the **Errors** menu item within the **Filter** menu, you will now enable filtering within the error monitoring window for that application. If you rerun the application, the errors that you selected for filtering will have been prevented from being displayed within the window when that particular error is received by ***ViewPort***.

To better understand how to select errors for filtering, you should understand the method that ***ViewPort*** uses when determining if an error received should be displayed. ***ViewPort*** uses the basic information of the error value (this includes parameter errors), the API for which the error occurred, the filename from which the API call was made along with the line number, API return value, and array index if relevant.

Therefore, when selecting an error for inclusion, you do not need to be repeatedly select the same error which appears more than once within the error monitoring window when it is a result of the same code being executed at different times. Now if the code is executed in different threads, then you would need to select it two or more times.

Filter preloading

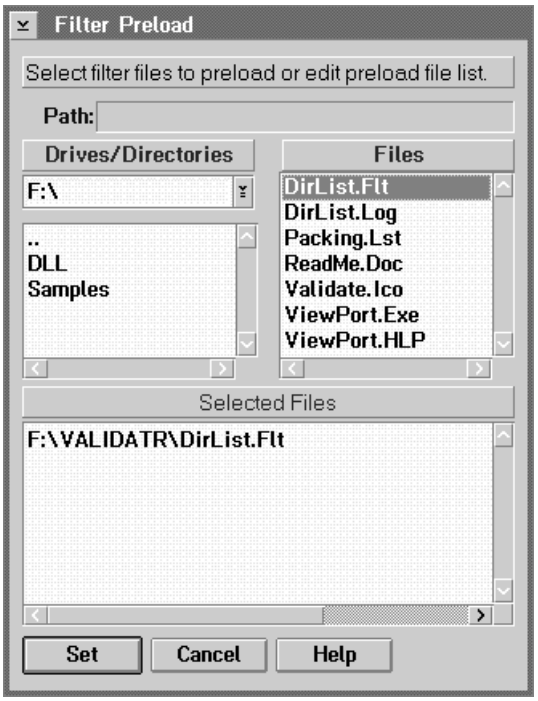
To maximize the usage of error filtering within ***ViewPort***, you can have ***ViewPort*** preload filter files that you have saved. This allows you to define for a given application a working set of errors that you expect and have the set loaded when ***ViewPort*** begins executing. Then when you

start the application that you are monitoring, the information will have already been preloaded and when the **ValInitialize** function is used within the application, ***ViewPort*** will check the application name against the windows that are filtering windows (the word [Filtered] will appear within the title bar) and if it finds a window with the same application name, the errors received from the validation routines will be directed to that window.

You define the filter files to preload through the **Filter Preload** dialogue. The dialogue contains two list boxes which allow you to select the drive, directory and file to use as the preload. It also contains a multiple-line entry field that allows you to enter the filter file names, change the locations of existing ones or to delete ones no longer required.

You select a drive or directory to change to by double clicking the mouse pointer on the entry within the **Drives/Directories** list box within the dialogue. Upon changing to that drive or directory, a list of files contained within that directory will be displayed. You can select the file to use by simply double-clicking on the entry within the **Files** list box causing the file to be added to the **Selected Files** multiple line entry field.

Plate 4.24
Filter Preload dialogue



If you want to quickly edit the location of a file listed within the **Selected Files** multiple line entry field, all you need to change is the appropriate part of the path for the file. To delete the entry, you only need to select the line containing the entry and then press the **Del** key on the keyboard.

The only restriction that applies to the **Selected Files** multiple line entry field is that each filter file entry must be placed on a separate line otherwise the full line will be treated as one entry. You cannot use tabs to separate different filter files.

When you click the mouse pointer on the **Set** push button, the entries within the **Selected Files** multiple line entry field will be placed within the preload filter files index such that the next time ***ViewPort*** is started, the filter files will be automatically loaded. If the list is empty, the preload filter files index will be purged such that ***ViewPort*** when next started will not preload any filter files.

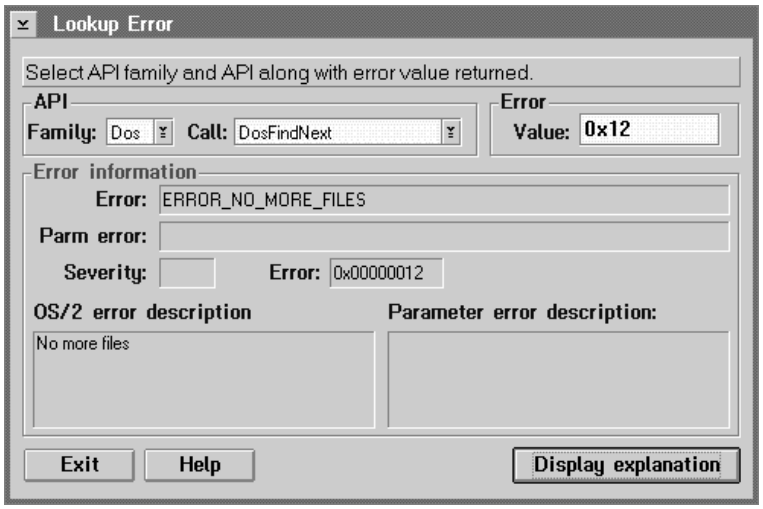
Error lookup

ViewPort provides you with a lookup facility where you can lookup the error symbol and error description for an error value you error for a given API along with an error symbol description lookup.

Lookup Error dialogue

The first is used primarily when you get an error value back from an API. The **Lookup Error** dialogue is used to select the API family, API and enter the error value such that when you request the explanation to be displayed, the corresponding error symbol and description.

Plate 4.25
Lookup Error dialogue



To be able to successfully lookup the error, you need to select the API family and call. The **Family** drop-down will contain the following family types:

Table 4.20
API family types

Designation	Family
Ddf	All calls prefixed by **Ddf**. These calls belong to the Dynamic Data Formatting API's.
Dev	All calls prefixed by **Dev**. These calls belong to the Device API's.
Drg	All calls prefixed by **Drg**. These calls belong to the Drag and Drop API's.
Dos	All calls prefixed by **Dos**. These calls belong to the kernel API's.
Gpi	All calls prefixed by **Gpi**. These calls belong to the Graphics Programming Interface API's.
Nls	All calls prefixed by **Nls**. These calls belong to the Nation Language Support API's. These calls at the present time cannot be monitored through ***Validator*** since they are still 16-bit.
Pic	All calls prefixed by **Pic**. These calls belong to the Picture Interchange API's.
Prf	All calls prefixed by Prf. These calls belong to the Profile API's.
Prt	All calls prefixed by **Prt**. These calls belong to the Print API's.
Spl	All calls prefixed by **Spl**. These calls belong to the Spooler API's.
Win	All calls prefixed by **Win**. These calls belong to the Window API's.
MMPM/2	All calls prefixed by **Spi**, **mci** and **mmio**. These calls belong to the MMPM/2 API's.
Pen	All calls prefixed by **Red**, **Vkp** and **Wrt**. These calls belong to the Pen for OS/2 API's.

When you select an entry from the **Family** drop-down, the drop-down labeled **Call** will be cleared of the current entries and replaced with the valid API's for the family selected.

You then select the API to use from that list. The error value is then entered within the entry field labeled **Value**. The number that you enter can be either decimal or hexadecimal. You can then click the mouse pointer on the **Display explanation** push button located in the bottom right corner of the dialogue.

The error symbol will be displayed within the field labeled **Error**. If the value that you entered was a parameter error, the corresponding parameter error symbol as defined within ValErr.H file would be displayed within the field labeled **Parm error**. If the error was from the Ddf*, Dev*, Drg*, Gpi*, Pic*, some Spl* and Win* families, the **Severity** field would show the severity level for the error. The levels that will be shown are: **Warning**, **Error**, **Severe**, and **Unrecoverable**.

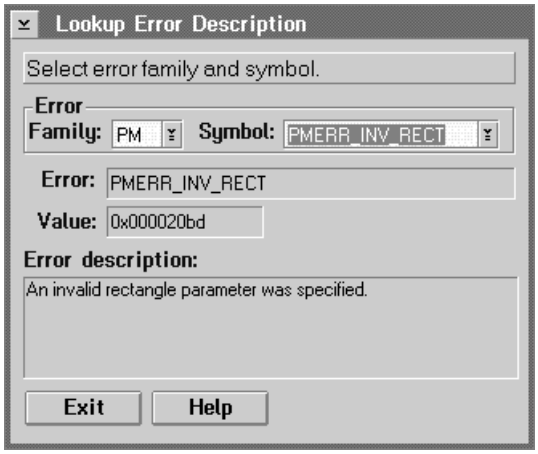
The hexadecimal value of the error is contained within the field labeled **Value**. Any OS/2 error description will be displayed within the field labeled **OS/2 error description** and if the error is a parameter error, the description for the parameter error is contained within the field labeled **Parameter error description**.

You can enter and display as many errors as you wish within this dialogue.

Lookup Error Description dialogue

When you know the error symbol but would like to see further information on the error, you can use the **Lookup Error Description** dialogue to retrieve an explanation of the error. The dialogue contains two drop-down lists that are used to select the error family and error symbol.

Plate 4.26
Lookup Error Description dialogue



The **Family** drop-down will contain the following family types:

Table 4.21
Error family types

Designation	Family
Base	All errors prefixed by ERROR_. These errors belong to the Dos*, Prt* and some Spl* API's.
Help	All errors prefixed by HMERR_. These errors are the result of the following API's: **WinAssociateHelpInstance**, **WinCreateHelpInstance**, **WinCreateHelpTable**, **WinDestroyHelpInstance**, **WinLoadHelpTable**, and **WinQueryHelpInstance**.
PM	All errors prefixed by PMERR_. These errors belong to the Ddf*, Dev*, Drg*, Gpi*, Pic*, Prf*, some Spl* and most Win* except those noted in the Help family API's.
MMPM/2	All errors prefixed by ERROR_, MCIERR_ AND MMIOERR_. These errors belong to the MultiMedia for PM/2 (MMPM/2) API's.
Pen	All errors prefixed by REDERR_, VKPERR_ and WRTERR_. These errors belong to the Pen for OS/2 API's.

When you select an entry from the **Family** drop-down, the drop-down labeled **Symbol** will be cleared of the current entries and replaced with the valid error designations for the family selected.

You then select the error symbol to search from that list upon which the error symbol will be placed into the field labeled **Error** and the value of the symbol will be placed in the field labeled **Value**. The description of the error will be placed in the field labeled **Error description**.

Interpreting Results

Introduction

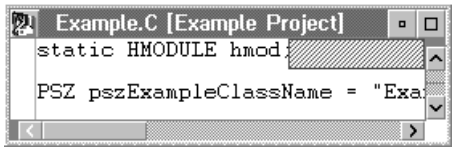
Even though the usage of the ***Validator*** tools is quite straight forward, and the error results usually are self apparent, you may want to look at the logic surrounding the call anyways since it may yield some subtle, yet effective changes in the logic of your application.

One aspect that is often forgotten when writing an application, you don't know how many thousands of lines of code may be executed when you use a system API. Therefore, if you don't need to use the system API and you have to add a few lines of code to perform a test on whether a specific API should be used, it may be worth while.

Basic logic example

Consider the following situation. You have to calculate a rectangle that will be filled with a particular colour. Plate 5.1 illustrates the problem.

Plate 5.1
Rectangle drawing zone



The basic logic is such that the text is first drawn within a rectangle and the remaining portion of the line is filled with the background colour. The following code fragment shows the manner in which this could be done:

Figure 5.1
Text drawing code

```
rcl.xLeft = 0L;
rcl.xRight = strlen(pszText) * fm.lAveCharWidth;

WinDrawText(hPS, -1L, pszText, &rcl,
            CLR_BLACK, CLR_WHITE, DT_ERASERECT | DT_LEFT | DT_TOP);

rcl.xLeft = rcl.xRight + 1L;
rcl.xRight = rclWindow.xRight;
WinFillRect(hPS, &rcl, CLR_WHITE);
```

As you can see, the code for the drawing is quite simple. It may look very innocent, but under certain circumstances, it will perform in a manner such that an error will be generated which could have been avoided.

For the first line shown in Plate 5.1, you will notice that the text is only filling up part of the line. The remaining portion of the line, which is shaded here to illustrate the rectangle area, would then be filled with the white background colour. For this line, there would be no problems. It is the next line that causes the error to be generated.

You will notice that the second line completely fills the window width. Therefore when the rectangle for the fill is calculated, the *xLeft* will be greater than *xRight*. ***Validator*** through its validation routines will detect this problem and issue an error PMERR_INV_RECT with a parameter error of PERR_WFR02_XRIGHTLTXLEFT.

This may seem very innocent, but you have to remember that no fill will be done and who knows how many lines of code have to be executed by the **WinFillRect** API before it determines that it has nothing to do and that the rectangle is in error.

The revised code would look something like:

Figure 5.2
Revised text drawing code

```
rcl.xLeft = 0L;
rcl.xRight = strlen(pszText) * fm.lAveCharWidth;

WinDrawText(hPS, -1L, pszText, &rcl,
            CLR_BLACK, CLR_WHITE, DT_ERASERECT | DT_LEFT | DT_TOP);

if ( (rcl.xLeft = rcl.xRight + 1L) <= rclWindow.xRight )
    {
    rcl.xRight = rclWindow.xRight;
    WinFillRect(hPS, &rcl, CLR_WHITE);
    }
```

By adding one line of code, you have effectively sped up your application under certain conditions. What may seem like an extra line of code may in fact act as a sentinel preventing you from executing hundreds or thousands of lines of code.

Repetitive search example

Another example is where you expect an error return to indicate an end of condition. A good example of this is the **DosFindFirst/FindNext** pair of API's. The **DosFindNext** API is expected to return the error code ERROR_NO_MORE_FILES to indicate the directory search is complete.

By closely looking at both of the API's, you will see that the API has been designed to allow for multiple entry requests where more than one directory entry can be retrieved at one time. Therefore, by looking at the normal code first and seeing that **DosFindNext** returns an error

indicating the end of the search, is there another method of achieving the same result but perhaps not using the **DosFindNext** API?
Starting with the following example code, you can see how this can be done.

Figure 5.3
Directory retrieval example - typical coding

```
                        /* Start directory search                        */
if ( DosFindFirst("*.*", &hDir, MUST_HAVE_DIRECTORY | FILE_DIRECTORY,
                  (PVOID)&findbuf, sizeof(FILEFINDBUF3),
                  &ulFileCnt, FIL_STANDARD) )
   {
   do
                        /* If file found not a sub-directory, place      */
                        /* filename in list box                          */

      if ( ((findbuf.attrFile & FILE_DIRECTORY) == FILE_DIRECTORY) &&
           strcmp(findbuf.achName, ".") )
          {
          sprintf(szStrBuf, "[%s]", findbuf.achName);
          WinSendMsg(hwndDir, LM_INSERTITEM,
                     MPFROMSHORT(LIT_SORTASCENDING),
                     MPFROMP(szStrBuf));
          }
                        /* Search for remaining entries and place valid  */
                        /* entries in list box                           */

       while ( !DosFindNext(hDir, &findbuf, sizeof(FILEFINDBUF3), &ulFileCnt) );

                        /* Close directory search handle                 */
   DosFindClose(hDir);
   }
```

First, the **DosFindFirst** API is used to start the directory search. The value contained within the *ulFileCnt* is 1 and the buffer used for the directory data is sized for one entry. The routine takes the values returned by the API and places the formatted results within a list box. It then at the bottom of the **do while** loop gets the next directory entry using the **DosFindNext** API and repeats the process. This loop continues until **DosFindNext** returns a non-zero value which will most likely be ERROR_NO_MORE_FILES.

The logic to the search is very straight-forward and easy to follow. It seems to be compact and fast. But again, you have to remember the number of times you go into the **DosFindNext** API. If, as an example, the directory contains 86 entries, **DosFindNext** would be used 86 times. In 85 of the calls, it would return one directory entry and in the last call made, it would return with an error code.

The call itself is defined as allowing for a buffer that can contain more than one entry and that you can request through both the **DosFindFirst** and **DosFindNext** API's the number of entries that should be returned if found. Therefore, what you can do is minimize the number of times the **DosFindNext** call is used while optimizing the number of entries that is returned by either API.

Figure 5.4 shows the revised code such an implementation:

Figure 5.4
Directory retrieval example
- optimized coding

```
DosAllocMem((PPVOID)(PVOID)&pfindbuf3, 64UL * sizeof(FILEFINDBUF3),
             PAG_READ | PAG_WRITE | PAG_COMMIT);
if ( DosFindFirst("*.*", &hDir, MUST_HAVE_DIRECTORY,
                  (PVOID)pfindbuf3, 64UL * sizeof(FILEFINDBUF3),
                  &ulFileCnt, FIL_STANDARD)) )
   {
   do
       for ( i = 0, pfindbuf = pfindbuf3; i < ulFileCnt; i++ )
           {
                       /* If file found not a sub-directory, place      */
                       /* filename in list box                          */

           if ( ((pfindbuf->attrFile & FILE_DIRECTORY) == FILE_DIRECTORY) &&
                strcmp(pfindbuf->achName, ".") )
               {
               sprintf(szStrBuf, "[%s]", pfindbuf->achName);
               WinSendMsg(hwndDir, LM_INSERTITEM,
                          MPFROMSHORT(LIT_SORTASCENDING),
                          MPFROMP(szStrBuf));
               }
           pfindbuf = (PFILEFINDBUF3)((PBYTE)pfindbuf +
                                      pfindbuf->oNextEntryOffset);
           }
                       /* Search for remaining entries and place valid  */
                       /* entries in list box                           */

       while ( (ulFileCnt == 64UL) &&
               !DosFindNext(hDir, pfindbuf3, 64UL * sizeof(FILEFINDBUF3),
                            &ulFileCnt) );

                       /* Close directory search handle                 */
   DosFindClose(hDir);
   }
DosFreeMem((PVOID)pfindbuf3);
```

A buffer is allocated that is large enough to contain 64 entries. The **DosFindFirst** call is given the address of the buffer and the *ulFileCnt* variable is set to 64 before entry thereby instructing **DosFindFirst** to return up to 64 directory entries if possible. The logic between the **DosFindFirst** and the **DosFindNext** has been changed to loop through the returned directory list and placing each entry found within the list box. Once all of the entries found have been placed within the list box, the returned count is checked within the **while** component of the **do** loop to see if 64 entries were returned. When the *ulFileCnt* is 64, there is a very good likelihood that there are more entries within the directory. If the value within *ulFileCnt* is less than 64, there is no need to perform the **DosFindNext** since all it will do is return an error code.

Using the example outlined above where there are 86 directory entries, **DosFindFirst** would be called and it would return 64 entries. These entries would then be processed and then, since the return count was 64, **DosFindNext** would be called to try to get another 64 entries. This next batch of entries would be processed after which the return count, 22, would be seen not to equal the 64 and the **do while** loop would be exited.

In this case, **DosFindNext** was only called once instead of 86 times as before. It can only be guessed as to the number of lines of code not executed in this case, but no matter, there is a savings in not having to perform the calls which do have an associated overhead.

Viewing results

So, you may be asking how does this relate back to observing the results of the validation routines? Simply, if you have a set of code that is always being executed and is somewhere in its logic returning an error, you may want to look at that code closely to determine if it can be optimized such that the error is not returned, or the call is not made where it will return the error code.

There are cases where you will always get an error code and there is really nothing that you can do about it. A good example is the usage of the **PrfQuery*** calls where you allow the user to configure the application to their own preferences. If the user has not configured a particular area, the **Prf*** call will always return an error code simply because no value has been recorded, and because of program design, is perfectly permissible. Figure 5.5 is such an example:

Figure 5.5 **Prf*** example

```
cbBuf = sizeof(BOOL);
if ( !PrfQueryProfileData(HINI_USER, pszProgramName, pszLookupKey,
                          &fViewOnly, &cbBuf) )
    fViewOnly = FALSE;
```

When the key value for **PrfQueryProfileData** is not found, the return value of the API is zero (0) indicating an error condition. By performing a **WinGetLastError**, the error returned by the call is PMERR_NOT_IN_IDX. Unless you implement the values recorded as a structure that is always recorded within the OS2.INI file, there is no way of getting around the error return. This is the only way that the application can determine if the values have been recorded.

Another problem that you may encounter is where visually the logic of the call may appear to be correct but a subtle problem is contained within the call. The following is such an example:

Figure 5.6 **DosFreeMem** example

```
DosFreeMem((PVOID)&pMem);
```

The call looks okay but it will indeed return an error. You may think that you have freed the memory earlier but that is not really the problem.

The problem is a result of the address of (&) operator. **DosFreeMem** is defined to have a PVOID parameter. The root of the problem would most likely have been due to the usage of the **DosAllocMem** as the template for the **DosFreeMem** where the line where **DosAllocMem** was used was copied and pasted before being converted to **DosFreeMem**. The only thing that was not adjusted properly was &.

The validation routines would correctly determine the problem and report the error, but, you still have to be able to look at the error returned

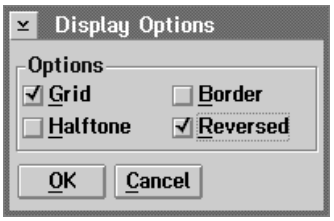
with the actual call itself sometimes to determine exactly where the problem is.

Adapting applications

An area where ***Validator*** is very useful is when you are adapting an existing application for another use or for a functional sub-set and you are removing controls from dialogues and windows. Because of the design of OS/2 Presentation Manager, you communicate with controls and windows using a message based mechanism. Also, each control has a handle which is used by the system to identify it.

For example, consider Plate 5.2 which is a dialogue containing four check boxes.

Plate 5.2
Original dialogue



The normal logic for the setting the check boxes appears in Figure 5.7

Figure 5.7
Original check box code

```
WinSendDlgItemMsg(hWnd, CB_GRID,     BM_SETCHECK, MPFROMSHORT(fGrid),     0L);
WinSendDlgItemMsg(hWnd, CB_HALFTONE, BM_SETCHECK, MPFROMSHORT(fHalftone), 0L);
WinSendDlgItemMsg(hWnd, CB_BORDER,   BM_SETCHECK, MPFROMSHORT(fBorder),   0L);
WinSendDlgItemMsg(hWnd, CB_REVERSED, BM_SETCHECK, MPFROMSHORT(fReversed), 0L);
```

When the dialogue is being adapted, the Reversed check box is removed such that the revised dialogue appears like that in Figure 5.3.

Plate 5.3
Revised dialogue

If the original code in Figure 5.7 is used with this new dialogue, Validator will determine that there is no corresponding control for CB_REVERSED and will denote an error for the function where the parameter error issued will be PERR_WSDIM02_ITEMDOESNOTEXIST. This will help you to determine where you have none functional code in the source code you are adapting.

Technical Requirements

Computer:	Computer capable of running OS/2 2.x
Numeric Co-processor:	Not required
Pointing Device:	Required
Minimum Memory:	4 MBytes
Recommended Memory:	8 MBytes
Minimum Disk Drives:	1 Diskette Drive and Hard Disk Drive (Required)
Video Adapters:	VGA Super-VGA IBM 8514/A IBM XGA
Operating System:	OS/2 2.x Series: **IBM Operating System/2** Version 2.x. OS/2 2.1 is highly recommended due to improvements made in version like in debugging support.
Disk space required:	3.8 MBytes

System Calling Conventions

The system calling convention that is used by the OS/2 API's is defined as the following:

- Parameters are passed on the 80386 stack.
- Parameters are pushed onto the stack using the C parameter passing convention of right to left order.
- The parameters are removed from the stack after the call has returned by the calling function.
- Parameters are double word (4-byte) aligned.
- **The parameter list size is passed in AL.**
- All functions returning non-floating-point values pass a return value back to the caller in EAX.

This affects the manner in which ***Validator*** utilizes the features as designed by OS/2 and implemented by the various compiler vendors. The implication that the number of parameters placed on the stack can be validated is useful. The only problem is that only one compiler vendor to date has adhered to the calling convention. All of the compilers except for the IBM C Set/2 and IBM C Set++ compilers do not completely follow the system calling convention. The only area that they do not adhere to is the placing within the AL register the number of parameters push on the stack.

Because of this, ***Validator*** provides two sets of validation DLL's: one set that is used by the IBM compilers which can validate the number of

parameters on the stack; and, one set for the compilers which do not place the number of parameters on the AL register.

Consider the following source code in Figure B.1. The code is used to call the OS/2 API **DosOpen**.

Figure B.1
C source code

```
VOID OpenFile( )

{
HFILE hFile;                    /* File Handle Holder                    */
ULONG ulAction;                 /* Action Taken Holder                   */

DosOpen("Filename.Ext", &hFile, &ulAction, 0UL,
        FILE_HIDDEN | FILE_NORMAL | FILE_ARCHIVED,
        OPEN_ACTION_OPEN_IF_EXISTS | OPEN_ACTION_FAIL_IF_NEW,
        OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
        NULL);
}
```

The assembler code in Figure B.2 is similar to that produced by the IBM C Set++ compiler. You will notice that the instruction **MOV AL,08H** is used to indicate to the calling function the number of parameters contained on the stack.

Figure B.2
Resultant assembler source code produced by the IBM C Set++ compiler

```
          ASSUME  CS:FLAT, DS:FLAT, SS:FLAT, ES:FLAT
          EXTRN   DosOpen:PROC
DATA32    SEGMENT
@STAT1    DB      "Filename.Ext",0H
DATA32    ENDS

CODE32    SEGMENT

;***** 11 VOID OpenFile( )
;***** 13
          ALIGN   010H

          PUBLIC  OpenFile
OpenFile  PROC
          SUB     ESP,08H           ; Stack allocation for hFile and ulAction

;***** 17 DosOpen("Filename.Ext", &hFile, &ulAction, 0UL,
;***** 18       FILE_HIDDEN | FILE_NORMAL | FILE_ARCHIVED,
;***** 19       OPEN_ACTION_OPEN_IF_EXISTS | OPEN_ACTION_FAIL_IF_NEW,
;***** 20       OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
;***** 21       NULL);
          MOV     AL,08H            ; Parameter Count
          PUSH    0H                ; NULL
          PUSH    042H              ; OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE
          PUSH    01H               ; OPEN_ACTION_OPEN_IF_EXISTS |
                                    ; OPEN_ACTION_FAIL_IF_NEW
          PUSH    022H              ; FILE_HIDDEN | FILE_NORMAL | FILE_ARCHIVED
          PUSH    0H                ; 0UL
          LEA     EDX,[ESP+04H]     ; ulAction
          PUSH    EDX
          LEA     ECX,[ESP+0cH]     ; hFile
          PUSH    ECX
          PUSH    OFFSET FLAT:@STAT1 ; "Filename.Ext"
          CALL    DosOpen
;***** 22 }
          ADD     ESP,028H
          RET
OpenFile  ENDP
CODE32    ENDS
          END
```

The assembler code in Figure B.3 is similar to that produced by the IBM C Set++ compiler except that it is produced by the WATCOM C/C++[32] compiler. You will notice that the instruction **MOV AL,08H** is not used to indicate to the calling function the number of parameters contained on the stack. Therefore, the validation routines cannot utilize this feature of

the system calling convention to verify that the correct number of parameters have been placed on the stack for the call.

Figure B.3
Resultant assembler source code produced by the WATCOM C/C++[32] compiler

```
            EXTRN   DosOpen :BYTE

CONST       SEGMENT DWORD PUBLIC USE32 'DATA'
L1          DB      46H,69H,6cH,65H,6eH,61H,6dH,65H
            DB      2eH,45H,78H,74H,00H
CONST       ENDS

_TEXT       SEGMENT BYTE PUBLIC USE32 'CODE'
            ASSUME  CS:_TEXT ,DS:DGROUP,SS:DGROUP
            PUBLIC  OpenFile
OpenFile: SUB     esp,00000008H      ; Stack allocation for hFile and ulAction

;  DosOpen("Filename.Ext", &hFile, &ulAction, 0UL,
;           FILE_HIDDEN | FILE_NORMAL | FILE_ARCHIVED,
;           OPEN_ACTION_OPEN_IF_EXISTS | OPEN_ACTION_FAIL_IF_NEW,
;           OPEN_ACCESS_READWRITE |  OPEN_SHARE_DENYNONE,
;           NULL);

            PUSH    00000000H          ; NULL
            PUSH    00000042H          ; OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE
            PUSH    00000001H          ; OPEN_ACTION_OPEN_IF_EXISTS |
                                       ; OPEN_ACTION_FAIL_IF_NEW
            PUSH    00000022H          ; FILE_HIDDEN | FILE_NORMAL | FILE_ARCHIVED
            PUSH    00000000H          ; 0UL
            LEA     eax,+14H[esp]      ; ulAction
            PUSH    eax
            LEA     eax,+1cH[esp]      ; hFile
            PUSH    eax
            PUSH    OFFSET DGROUP:L1   : "Filename.Ext"
            CALL    NEAR PTR DosOpen
            ADD     esp,00000020H
            ADD     esp,00000008H
            RET
_TEXT       ENDS
            END
```

W

Z

Validator was developed using IBM C Set++ for OS/2 Version 2.1 on IBM OS/2 Version 2.11 with an IBM Model 90-0KD.

This manual was prepared using Microsoft *Word for Windows* Version 2.0c on IBM OS/2 Version 2.11 with an IBM Model 90-0KD. Proofs were printed on an IBM LaserPrinter 4039 12R with 12 MBytes of memory and the Enhanced PostScript Option.