Overview of the Build Process

This guide describes how to build 32-bit applications for Windows and dynamic-link libraries (DLLs) using the Microsoft Win32 Software Development Kit (SDK). It describes the components of a generalized makefile and includes information on using the C run-time libraries.

The following table summarizes the steps used to build a Win32 application or DLL:

Steps to Building Applications

Step	Tool
1. Compile C and C++ source-language files into object files.	C/C++ compiler (CL)
2. Create and edit resources. Doing so may also create include (.H) files which define useful constants.	Dialog Editor, Image Editor, and Font Editor (DLGEDIT, IMAGEDIT, and FONTEDIT)
3. Compile resource scripts to linkable resource files.	Resource Compiler (RC)
4. Compile the module-definition file for each DLL to an import library and export library.	Library manager (LIB)
5. Link the object modules, resources, standard libraries, and import libraries (for an application using DLLs) or export library (for a DLL) to produce an application.	Linker (LINK)
6. Use the appropriate switches to build a debugging version of the application or DLL.	Linker switches: **/debug:full**, **/debugtype:cv**

If you are familiar with the process of building applications and DLLs for Windows 3.*x*, you will notice some differences. The following parts of the build process are new or different with Win32:

- Resources are linked along with object modules and libraries. You do not need to run the resource compiler to add resources to the executable file.
- When building a DLL, create an import library from a .DEF file, then link the DLL with the import library. The linker does not accept a .DEF file when resolving imports.
- When linking a DLL, you must specify the name of the initialization routine using the linker **/ENTRY** option. This is the result of the new DLL initialization and termination model in Win32.
- You link a DLL with an export library (.EXP file). The export library is generated by LIB at the same time it generates the import library from the .DEF file.

The sample makefiles provided with the SDK samples give good examples of the build process. Each of them includes WIN32.MAK, which defines most of the common macros you need to build 32-bit applications for Windows NT and Windows 95. For information on source code considerations in porting your code from 16- to 32-bits, see the following topics:

Porting 16-Bit Code to 32-Bit Windows
Handling Messages with Portable Macros
Writing Portable C Programs
WINDOWS.H and STRICT Type Checking

Using WIN32.MAK

It is recommended that you examine the contents of WIN32.MAK, located in the \MSTOOLS\H subdirectory. There are macros defined in this makefile template that can be used to simplify your own makefiles and to ensure that they are properly built to avoid conflicts.

For example, WIN32.MAK includes the following macros to simplify compilation:

- **$(CVARSDLL)**
  This macro is used for DLLs (single-threaded or multi-threaded). It expands to the following flags:
  ```
  -DWIN32 -D_WIN32 -DNULL=0 -D_MT -D_DLL
  ```

- **$(CVARS)**
  This macro is used for single-threaded executables. It expands to the following flags:
  ```
  -DWIN32 -D_WIN32 -DNULL=0
  ```

WIN32.MAK also includes macros to simplify linking, among which are the following:

- **$(CONLIBSDLL)**
  This macro is for console DLLs using CRTDLL.LIB. It expands to the following list of libraries:
  ```
  CRTDLL.LIB KERNEL32.LIB ADVAPI32.LIB
  ```

- **$(GUILIBSDLL)**
  This macro is used for GUI DLLs using CRTDLL.LIB. It expands to the following list of libraries:
  ```
  CRTDLL.LIB KERNEL32.LIB ADVAPI32.LIB USER32.LIB GDI32.LIB \
  COMDLG32.LIB WINSPOOL.LIB
  ```

Building Applications

In the following example, the object files and libraries are linked to produce the application executable file. Note that the resource file (*.res) is linked along with the object files.

The following is a simple example of a linker command line (used in an inline response file):

```
#  Build an executable (.exe) file
$(TARGET).exe : $(OBJS) $(TARGET).res
     $(LINK) @<<
/out:$(TARGET).exe
/debug:notmapped,full
/debugtype:cv
/machine:$(CPU)
/subsystem:windows
$(LINK_FLAGS)
$(OBJS)
$(TARGET).res
$(LIBS)
<<
```

Building DLLs

The following example illustrates how to create the import and export libraries for a DLL. The import library is linked with the application that uses this DLL. The export library is linked with the DLL. The export library is not explicitly listed in the command because the library tool automatically generates it along with the import library.

```
#  Generate import library (.lib) and export library (.exp)
#  from a module-definition (.def) file for a DLL
$(TARGET).lib $(TARGET).exp : $(TARGET).def
     $(IMPLIB) /out:$(TARGET).lib /machine:$(CPU)
     /def:$(TARGET).def
```

The following example links the DLL. The **/DLL** and **/ENTRY** options are required to build a DLL. The **/DLL** option specifies that the output file is a DLL. The **/ENTRY** option specifies the name of the DLL initialization routine.

```
# Build DLL using objects and export library
$(TARGET).dll : $(OBJS) $(TARGET).exp
     $(LINK) @<<
/out:$(TARGET).dll
/dll
-entry:_DllMainCRTStartup$(DLLENTRY)
/debug:full
/debugtype:cv
/machine:$(CPU)
/subsystem:windows
$(LFLAGS)
$(OBJS)
$(TARGET).exp
$(CONLIBSDLL)
<<
```

The WIN32.MAK file supplied in \MSTOOLS\H will help simplify the build process. The compiler and linker switches shown in the above examples are defined as macros in this file.

Using the C Run-Time Library

This guide describes how to use the different forms of the C run-time libraries when building your 32-bit application. It also describes how to specify the entry point when linking a DLL with the run-time libraries.

With the original Win32 SDK, three forms of the C run-time library were provided:

LIBC.LIB
Statically linked library for single-threaded applications.

LIBCMT.LIB
Statically linked library that supports multi-threaded applications.

CRTDLL.LIB
Import library for CRTDLL.DLL (the C run-time DLL) that also supports multi-threaded applications.

The C run-time library is not intrinsic to the Win32 SDK, however. These libraries should come from the same source as the compiler. For new computer architectures, the Win32 SDK typically provided the compiler and C run-time libraries until there were other offerings available. At that point, the SDK stopped including these pieces and allowed programmers to choose their favorite vendor.

The names of the libraries may differ from vendor to vendor. These name differences can be encapsulated in WIN32.MAK. The Microsoft names used in the Visual C++ product are LIBC.LIB, LIBCMT.LIB, and MSVCRT.DLL.

Calling the C Run-Time Library from a DLL

When linking a DLL with any of the C run-time libraries, the entry point for the DLL must be the routine _CRT_INIT, or your initialization code must explicitly call _CRT_INIT every time the DLL entry point is called.

{ewl msdncd, EWGraphic, group10225 0 /a "SDK.BMP"} To call _CRT_INIT if you do not have your own DLL entry point

1. Specify _CRT_INIT as the entry point of the DLL.
2. Assuming that you have included WIN32.MAK (which defines the macro DLLENTRY as @12), add the following option to the DLL's linker command line:

```
-entry:_CRT_INIT$(DLLENTRY)
```

{ewl msdncd, EWGraphic, group10225 1 /a "SDK.BMP"} To call _CRT_INIT if you have your own DLL entry point

1. Add the following code in the entry point, using this prototype for _CRT_INIT:

```
BOOL WINAPI _CRT_INIT( HINSTANCE hinstDLL, DWORD fdwReason,
   LPVOID lpReserved );
```

   For information on _CRT_INIT return values, see **DllEntryPoint**. This function returns the same values as _CRT_INIT.
2. On PROCESS_ATTACH and THREAD_ATTACH (see **DllEntryPoint** for more information on these flags), call _CRT_INIT at the beginning of the initialization routine, before any C run-time functions are called or any floating-point operations are performed.
3. Call your own process/thread initialization/termination code.
4. On PROCESS_DETACH and THREAD_DETACH, call _CRT_INIT near the end of the initialization routine, after all C run-time functions are called and all floating-point operations are completed.

Be sure to pass all of the parameters of the entry point to _CRT_INIT. Because _CRT_INIT expects these parameters, your application may not work reliably if they are omitted (in particular, fdwReason is required to determine whether process initialization or termination is needed).

The following is a sample entry point function that shows how and when to make these calls to _CRT_INIT in the DLL entry point:

```
BOOL WINAPI DllEntryPoint( HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpReserved )
{

     // Initialize the C run-time before calling any of your code.
     if( fdwReason == DLL_PROCESS_ATTACH || fdwReason ==
     DLL_THREAD_ATTACH )
           if( !_CRT_INIT( hinstDLL, fdwReason, lpReserved ) )
     return( FALSE );

     // Place your DLL's initialization/termination code here. If you
     // are porting from 16-bits and have an existing LibMain routine,
     // you can call it here. For example:

     // if( fdwReason == DLL_PROCESS_ATTACH )
     //          LibMain();

     // Terminate the C run-time after all your code.
```

```
    if( fdwReason == DLL_PROCESS_DETACH || fdwReason ==
    DLL_THREAD_DETACH )
          if( !_CRT_INIT( hinstDLL, fdwReason, lpReserved ) )
                return( FALSE );
          return( TRUE );

}
```

For more information, see **DllMainCRTStartup**.

DllMainCRTStartup

The complexity involved in initializing the C run-time library from a DLL is cleared up by using the **$(dllflags)** macro in NTWIN32.MAK. It will specify the following switch to the linker:

```
-entry:_DllMainCRTStattup$(DLLENTRY)
```

This entry point is exported from the C run-time libraries.

DllMainCRTStartup will call CRT_INIT and it will call the **DllMain** function exported from your own DLL. The order of the calls is dependent on whether the process or thread is attaching (in which case CRT_INIT is called first) or detaching (in which case CRT_INIT is called last).

In summary, export **DllMain** from your DLL, use the **$(dllflags)** macro in the link line of your makefile, and don't call CRT_INIT explicitly.

Using Multiple C Run-Time Libraries

If your application makes C run-time library calls and also calls functions contained in a DLL that makes C run-time library calls, note the following. If the executable and the DLL are both linked with one of the statically linked C run-time libraries (LIBC.LIB or LIBCMT.LIB), the executable and the DLL will have separate copies of all C run-time functions and global variables. This means that C run-time data cannot be shared between the executable and the DLL.

To avoid this problem, link both the executable and the DLL with CRTDLL.LIB. This allows both the executable and the DLL to use the common set of functions and data contained in CRTDLL.DLL. C run-time data such as stream handles can then be shared by both the executable and the DLL.

Mixing Library Types

If your DLL is linked with CRTDLL.LIB, any executables that call your DLL must also be linked with CRTDLL.LIB. Linking the executable with either LIBC.LIB or LIBCMT.LIB and the DLL with CRTDLL.LIB can cause unpredictable results. Unless you are sure that the DLL will only be called by EXEs that are linked with CRTDLL.LIB, you must link the DLL with one of the statically linked C run-time libraries (LIBC.LIB or LIBCMT.LIB). If you are unsure, you should link the EXE and all DLLs it will call with CRTDLL.DLL.

If a DLL is linked with LIBC.LIB, and the DLL may be called by a multi-threaded application, multiple threads running in this DLL at the same time will not be supported. This may cause unpredictable results. Therefore, if there is a possibility that the DLL will be called by multi-threaded programs, be sure to link it with one of the libraries that support multi-threaded programs (LIBCMT.LIB or CRTDLL.LIB).

Using Calling Conventions

The Intel®-based Windows C/C++ compilers provide several ways to call internal and external functions. The information in this guide can help you when debugging your program and when interfacing your code with assembly-language routines.

This guide covers the differences between the calling conventions, how arguments are passed, and how values are returned by functions.

Details on Calling Conventions

All arguments are widened to 32 bits when they are passed. Parameters are pushed onto the stack from right to left. Return values are also widened to 32 bits.

For Intel-based Windows C/C++ compilers, the return value is placed in the EAX register, except for 8-byte structures, which are returned in the EDX:EAX register pair. Larger structures are returned in the EAX register as pointers to hidden return structures.

For Intel-based Windows C/C++ compilers, the compiler generates prolog and epilog code to save and restore the ESI, EDI, EBX, and EBP registers, if they are used in the function.

Note For information on how to define your own function prolog and epilog code, see Storage Class Attributes.

Obsolete Calling Conventions

The **__pascal**, **__fortran**, and **__syscall** calling conventions are no longer supported. Their functionality can be emulated by using one of the supported calling conventions and appropriate linker options.

WINDOWS.H now supports the **WINAPI** macro, which translates to the appropriate calling convention for the target. Use **WINAPI** where you previously used **PASCAL** or **__far __pascal**.

Argument Passing and Naming Conventions

The following table lists the calling conventions supported by most Windows C/C++ compilers:

Supported Calling Conventions

Keyword	Stack clean-up	Argument passing
__cdecl	Caller	Pushed on stack
__stdcall	Callee	Pushed on stack
__fastcall	Callee	Stored in registers, then pushed on stack
thiscall (not a keyword)	Callee	Pushed on stack

For an example of these calling conventions, see Example.

__cdecl

This is the default calling convention for C and C++ programs. Because the stack is cleaned up by the caller, it can do **vararg** functions. The **__cdecl** calling convention creates larger executables than **__stdcall** because it requires each function call to include stack clean-up code.

All function arguments are pushed on the stack. In C, the **__cdecl** function names are prefixed by an underscore when decorated.

Note See your C++ compiler documentation for information on C++ name decoration.

__stdcall

The **__stdcall** calling convention is used to call Win32 API functions. The stack is cleaned up by the callee, so the compiler makes **vararg** functions **__cdecl**. Functions that use this calling convention require a function prototype.

All function arguments are pushed on the stack. In C, the **__stdcall** function names are prefixed by an underscore and suffixed with ***@number*** when decorated, where *number* is the number of bytes (in decimal) used by the widened arguments pushed to the stack.

__fastcall

The **__fastcall** calling convention is similar to the **__stdcall** calling convention, but **__fastcall** speeds up function calling by storing some number of arguments in registers. Any additional arguments are pushed to the stack as in the other calling conventions.

In C, the **__fastcall** function names are prefixed by the at sign (**@**) and suffixed with ***@number*** when decorated, where *number* is the number of bytes (in decimal) used by the widened arguments pushed to the stack plus the number of bytes taken by the widened arguments passed in registers.

Note Future compiler versions may use different registers to store parameters.

thiscall

This is the default calling convention used by C++ member functions that do not use variable arguments. The stack is cleaned up by the callee, so the compiler makes **vararg** functions **__cdecl** and pushes the **this** pointer on the stack last. The *thiscall* calling convention cannot be explicitly specified in a program because *thiscall* is not a keyword.

All function arguments are pushed on the stack. Because this calling convention applies only to C++, there is no C name decoration scheme.

Example

The following example shows the results of making a function call using various calling conventions. This example is specific to an Intel-based Windows C/C++ compiler.

Function Prototype and Call

This example is based on the following function skeleton. Replace *calltype* with the appropriate calling convention. The C decorated names are listed; see your compiler documentation for the C++ decorated names.

```
void calltype MyFunc( char c, short s, int i, double f );
.
.
.
void MyFunc( char c, short s, int i, double f )
     {
     .
     .
     .
     }
.
.
.
MyFunc ('x', 12, 8192, 2.7183);
```

Results

The following lists display the stack and register contents using the **__cdecl** calling convention. The C decorated function name is "_MyFunc".

Address	Contents
ESP+0x14	2.7183
ESP+0x10	
ESP+0x0C	8192
ESP+0x08	12
ESP+0x04	x
ESP	Return address

Register	Contents
ECX	Not used
EDX	Not used

The following lists display the stack and register contents using the **__stdcall** and *thiscall* calling conventions. The C decorated name (for **__stdcall**, not for *thiscall*) is "_MyFunc@20".

Address	Contents
ESP+0x14	2.7183
ESP+0x1	

ESP+0x0C	8192
ESP+0x08	12
ESP+0x04	x
ESP	Return address

Register	**Contents**
ECX	**this** (*thiscall* only)
EDX	Not used

The following lists display the stack and register contents using the **__fastcall** calling convention. The C decorated name is "@MyFunc@20".

Address	**Contents**
ESP+0x0C	2.7183
ESP+0x08	
ESP+0x04	8192
ESP	Return address

Register	**Contents**
ECX	x
EDX	12

Porting 16-bit Code to 32-bit Windows

When you begin writing applications for 32-bit Windows, you probably already have existing applications for 16-bit Windows. Porting these applications is the quickest way to start producing 32-bit software.

This guide describes how to create a 32-bit version of an application written for Windows 3.x in C, as well as how to make the code portable *between* versions of Windows. Portable code can be recompiled as either a 16-bit application or a 32-bit application, by setting an option.

This guide includes the following topics:

Overview of 32-bit Programming

The Application Program Interface (API) for Win32 uses the flat 32-bit addressing mode, supports source code portability with RISC processors, and supports high-end capabilities such as security and true multitasking.

One of the major design goals of the 32-bit API was to minimize the impact on existing code, so that 16-bit applications can be adapted as easily as possible. However, some changes were mandated by the larger address space. Pointers are all 32 bits wide and no longer **near** or **far**, and your code cannot make assumptions based on segmented memory.

Items which have increased to 32 bits include the following:

- Pointers
- Window handles
- Handles to other objects, such as pens, brushes, and menus
- Graphics coordinates

These size differences are generally resolved in the header files (see WINDOWS.H) or by the C language, but some changes to source code are necessary. Because the different sizes cause some message parameters to be packed with information differently, you must rewrite code that handles these messages. The larger size of graphics coordinates also affects a number of function calls.

Some source-code changes are required because Win32 uses higher-level mechanisms for certain operations, such as file-system calls. These mechanisms make the 32-bit API adaptable to a wide number of platforms, and it supports powerful new features such as multiple threads of execution.

Although Windows 3.x and Win32 were designed to be as compatible as possible, you may need to look carefully over a large amount of source code. Where do you start? The top-down approach is recommended:

1. Compile the application for 32 bits, and note the errors generated by the compiler.
2. Replace complex procedures that are difficult to port, as well as procedures written in assembly language, with stub procedures. (These do nothing except return).
3. Fix errors in the main portion of the application, using the techniques described in this guide.
4. Fill in each of the stub procedures, one at a time, with portable code once the main portion of the application compiles and runs correctly.

Using PORTTOOL to Automate Porting

You can use the PORTTOOL utility (PORTTOOL.EXE) to help port applications more easily. This utility finds locations in your code, such as references to certain functions and messages, that are likely to need revision. You should use PORTTOOL in conjunction with the information in this guide.

PORTTOOL uses settings in the file PORT.INI to determine what items to look for. This file is based on the Summary of Function and Message Differences table. You may want to add the following setting to this file, to make sure that PORTTOOL can find the Help file on the Win32 API:

```
[PORTTOOL]
    WinHelp=c:\mstools\bin\win32.hlp
```

Run PORTTOOL and load a Windows 3.x source file. Select the SearchAPI option from the Search menu to search for occurrences of problematic functions and messages. When an occurrence of either is found, a dialog box appears specifying the message or function and briefly suggesting what change needs to be made. Although the porting tool is not intended to replace your primary editor, it does support basic editing capabilities (such as Cut, Paste, and Search).

Steps in Porting Applications

The next few topics describe general steps you should take each time you port a 16-bit application to Win32. If your application uses advanced techniques, such as manipulating the WIN.INI file, focus, and mouse capture, you may need to consult Special Considerations for Advanced Applications.

The basic steps to porting an application include rewriting the Windows procedure, replacing **WORD** by proper data types, handling 32-bit messages, and adjusting some of the function calls.

Revising the Window Procedure Declaration

The first step in porting a Windows 3.x application is to revise the declaration of the window procedure. The declaration of a window procedure for a Windows 3.x application is shown below. Note the use of **FAR PASCAL**, **unsigned**, and **WORD**, respectively, in the first three lines:

```
LONG FAR PASCAL MainWndProc( HWND hWnd,
     unsigned message,
     WORD wParam,
     LONG lParam)
```

To revise the declaration for Win32, replace the data types used in Windows 3.x (**FAR PASCAL** and **WORD**) with **CALLBACK** and **WPARAM**. The revised version is shown below.

```
LRESULT CALLBACK MainWndProc(HWND hWnd,
     UINT message,
     WPARAM wParam,
     LPARAM lParam)
```

FAR PASCAL, shown in the Windows 3.x version of this code, can be used to compile applications for Win32, but it will be defined as an empty string (and is thus ignored). The **CALLBACK** function type is recommended because the header files define it as appropriate for the compiler; currently, Win32 uses the **stdcall** calling convention, but future versions of the Win32 Development Kit may use different conventions to optimize for more advanced processors.

The following table summarizes the changes to the declaration noted in the above example:

Changes to the Window Procedure Declaration

Windows 3.x	Win32 (portable code)	Reason for changing
FAR PASCAL	**CALLBACK**	**CALLBACK** is guaranteed to use whatever calling convention is appropriate for Windows
unsigned	**UINT**	Meaning is the same, but **UINT** guarantees portability for future platforms
WORD	**WPARAM**	**WORD** is always 16 bits. **WPARAM** is portable.
LONG	**LPARAM**	Not required, but consistent with **WPARAM** convention

The most significant difference between the Windows 3.x declaration and the portable version involves the *wParam* parameter, which under Win32 grows to 32 bits in size.

The combination of a 32-bit *wParam* message parameter, along with the fact that addresses and handles grow to 32 bits, means that a number of messages must be repacked, as described in Handling 32-bit Messages.

Using Proper Data Types

16-bit Windows source code often uses the type **WORD** interchangeably with types such as **HWND** and **HANDLE**. For example, the typecast **(WORD)** might be used to cast a data type to a handle:

```
hWnd = (WORD) SendMessage( hWnd, WM_GETMDIACTIVATE, 0, 0 );
```

This code compiles Windows 3.x application correctly, because both the **WORD** type and handles are 16 bits. But the code produces errors when compiled for Win32, because handles (such as **HWND** types) grow to 32 bits while the **WORD** type is still 16 bits.

To write portable code, examine each occurrence of **WORD** casts and data definitions in your code, and revise as follows:

- If a variable or expression is to hold a handle, replace **WORD** with **HWND**, **HPEN**, **HINSTANCE**, or another handle type.
- If a variable or expression is a graphics coordinate or some other integer value that grows from 16 to 32 bits, replace **WORD** with **UINT**.
- Maintain use of the **WORD** type only if the data type needs to be 16 bits for all versions of Windows (usually because it is a function argument or structure member).
- If in doubt, check reference documentation.

UINT is defined to be the natural integer size for the target environment: either 16 or 32 bits. Thirty-two-bit operating environments, especially with RISC processors, actually handle 32-bit data more efficiently than 16-bit data.

In the portable version of the previous example, the **(WORD)** cast is replaced by **(HWND)**:

```
hWnd = (HWND) SendMessage( hWnd, WM_GETMDIACTIVATE, 0, 0 );
```

In general, use of the most specific type possible is recommended for writing more portable code. Avoid using a generic handle type such as **HANDLE** when you can. Try to use a more specific type such as **HPEN**. You should also define specific types for application-specific objects you might create.

Handling 32-bit Messages

When you move from Windows 3.x to Win32, the information packing changes for some messages. You can either:

- Revise the code so that it works only for the 32-bit version.
- Make the message-handling code portable, so that you can easily compile for either the 16-bit or 32-bit environment.

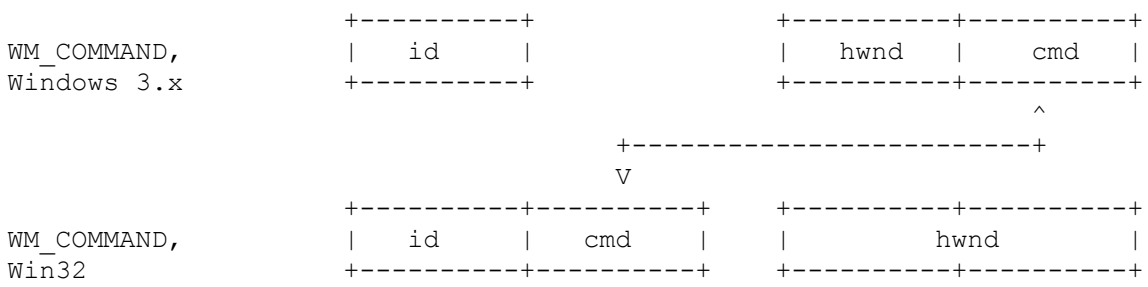
The second method is strongly recommended. This guide focuses primarily on writing portable versions of the code.

Handles grow to 32 bits under Win32 and can therefore no longer be combined with other information and still fit into a 32-bit parameter (*lParam*). The handle now occupies all of *lParam*, so information formerly in the high or low word of *lParam* must now move to *wParam*.

Because the *wParam* message parameter also grows to 32 bits, it can hold the information that can no longer be held in *lParam*. The following table shows how this repacking works for WM_COMMAND, one of the messages affected:

Environment	wParam	lParam
Windows 3.x	**id**	**hwnd**, **cmd**
Win32	**id**, **cmd**	**hwnd** (32 bits)

The following figure illustrates how the sizes of the parameters change, as well as how information is repacked:



Extracting Data from Messages with Portable Code

When your code handles a Windows message that has been repacked, the cleanest way to revise the code is to first extract needed information from the message and store the information in local variables. This localizes message packing issues to a few lines of your code.

For example, if your application was written for 16 bits, you can use the following code to handle the WM_COMMAND message:

```
            case WM_COMMAND:
                  id = wParam;
                  hwndChild = LOWORD(lParam);
                  cmd = HIWORD(lParam);
```

Here is the portable version of the same code, which yields correct results, whether you compile for the 16-bit or 32-bit environment:

```
            case WM_COMMAND:
                  id = LOWORD(wParam);
                  hwndChild = (HWND)(UINT)lParam;
#ifdef WIN32
                  cmd = HIWORD(wParam);
#else
                  cmd = HIWORD(lParam);
#endif
```

Each of the examples of data extraction above (**id**, **hwndChild**, and **cmd**) illustrates a common case of repacking problems. The following topics deal with each case in turn:

- Data always occupying the low 16 bits
- A handle changing size but not location
- Data that changes in location

Data Always Occupying the Low 16 Bits

The **id** data, in this case, always occupies the low word of *wParam* and is always 16 bits. The **LOWORD** macro produces the correct results because it always returns a 16-bit data type. The result is either all of *wParam* (if 16 bits) or the low half (if 32 bits):

```
      id = LOWORD(wParam);                  // 16-bit: id=wParam
```

Handles Not Changing Location

The **hwndChild** data is the same address as *lParam*; it is either in the low byte or it occupies all of *lParam*. As long as the address of a handle is always that of *lParam*, using the cast **(HWND)(UINT)** correctly extracts the handle:

```
      hwndChild = (HWND)(UINT)lParam;       // 16-bit: hwnd=LOWORD(lParam);
```

Data Changing Location

In cases where a piece of data changes packing location under Win32, you need to use an **#ifdef** statement (or you can write your own conversion macros). In this case, the cmd data moves from the high word of *wParam* to the high word of *wParam*, so the portable version of the code is:

```
#ifdef WIN32
      cmd = HIWORD(wParam);
#else
```

```
     cmd = HIWORD(lParam);
#endif
```

Often, **LOWORD** and **HIWORD** macros create portability problems, because you should usually (except as indicated here) not be extracting parts of data types.

Using Message Crackers to Write Portable Code

Message crackers are a set of macros that parse message parameters. They enable you to write portable code without having to use **#ifdef** statements to parse messages. Some examples of macros used to parse message parameters follow:

```
GET_WM_COMMAND_ID  (wParam, lParam)    // Parse control ID value
GET_WM_COMMAND_HWND(wParam, lParam)    // Parse control HWND
GET_WM_COMMAND_CMD (wParam, lParam)    // Parse notification command
```

For each version of Windows (Windows 3.x and Win32), there is a WINDOWSX.H header file that defines these macros as appropriate. Refer to Handling Messages with Portable Macros for more information.

Summary of Windows Messages Affected

Use the following table to reference the packing of Windows messages affected by porting. The approaches discussed in the previous sections can be used for each message. Where two parameters are given, the one listed first corresponds to the least-significant 16 bits.

Except for the WM_CTLCOLOR messages, each message is given below with two rows: the first row gives 16-bit Windows packing for the message, the one below it gives Win32 packing.

Message		wParam: Windows 3.x Win32	lParam: Windows 3.x Win32
WM_ACTIVATE	(16-bit Windows)	**state**	**fMinimized**, **hwnd**
	(Win32)	**state**, **fMinimized**	**hwnd** (32 bits)
WM_CHARTOITEM	(16-bit Windows)	**char**	**pos**, **hwnd**
	(Win32)	**char**, **pos**	**hwnd** (32 bits)
WM_COMMAND	(16-bit Windows)	**id**	**hwnd**, **cmd**
	(Win32)	**id**, **cmd**	**hwnd** (32 bits)
WM_CTLCOLOR	(16-bit Windows)	**hdc**	**hwnd**, **type**
WM_CTLCOLOR*type* (Win32)	(Win32)	**hdc** (32 bits)	**hwnd** (32 bits)

Note Under Win32, WM_CTLCOLOR is replaced by a series of messages, each corresponding to a different type. To write portable code, use **#ifdef** to handle this difference.

Message		wParam: Windows 3.x Win32	lParam: Windows 3.x Win32
WM_MENUSELECT	(16-bit Windows)	**cmd**	**flags**, **hMenu**
	(Win32)	**cmd**, **flags**	**hMenu** (32 bits)
WM_MDIACTIVATE	(16-bit Windows)	**fActivate**	**hwndDeactivate**, **hwndActivate**
	(Win32)	**hwndActivate** (32 bits)	**hwndDeactivate** (32 bits)
WM_MDISETMENU	(16-bit Windows)	0	**hMenuFrame**, **hMenuWindow**
	(Win32)	**hMenuFrame** (32 bits)	**hMenuWindow** (32 bits)
WM_MENUCHAR	(16-bit	**char**	**hMenu**,

	Windows)		**fMenu**
	(Win32)	**char**, **fMenu**	**hMenu** (32 bits)
WM_PARENTNOTIFY	(16-bit Windows)	**msg**	**id**, **hwndChild**
	(Win32)	**msg**, **id**	**hwndChild** (32 bits)
WM_VKEYTOITEM	(16-bit Windows)	**code**	**item**, **hwnd**
	(Win32)	**code**, **item**	**hwnd** (32 bits)
EM_GETSEL (returns wStart, wEnd)	(16-bit Windows)	0	0
	(Win32)	0 or **lpdwStart**	0 or **lpdwEnd**
EM_LINESCROLL	(16-bit Windows)	0	**nLinesVert**, **nLinesHorz**
	(Win32)	**mLinesHorz** (32 bits)	**nLinesVert** (32 bits)
EM_SETSEL	(16-bit Windows)	0	**wStart**, **wEnd**
	(Win32)	**wStart** (32 bits)	**wEnd** (32 bits)
WM_HSCROLL, WM_VSCROLL	(16-bit Windows)	**code**	**pos**, **hwnd**
	(Win32)	**code**, **pos**	**hwnd** (32 bits)

Summary of DDE Messages Affected

DDE messages are packed differently for Win32 and Windows 3.x. These differences are shown in the following table.

Message		wParam: Windows 3.x Win32	lParam: Windows 3.x Win32
WM_DDE_ACK (posted form only)	(16-bit Windows)	**hwnd**	**wStatus**, **aItem** or **wStatus**, **hCommands**
	(Win32)	**hwnd** (32 bits)	**hDDEAck** (see below)
WM_DDE_ADVISE	(16-bit Windows)	**hwnd**	**hOptions**, **aItem**
	(Win32)	**hwnd** (32 bits)	**hDDEAdvise** (see below)
WM_DDE_DATA	(16-bit Windows)	**hwnd**	**hData**, **aItem**
	(Win32)	**hwnd** (32 bits)	**hDDEData** (see below)
WM_DDE_POKE	(16-bit Windows)	**hwnd**	**hData**, **aItem**
	(Win32)	**hwnd** (32 bits)	**hDDEPoke** (see below)

Because of storage limitations, some of the information in the 32-bit versions of the messages is stored in a structure, which is accessed through the handle in *lParam*. You can use the following functions to extract information from these structures:

PackDDElParam
UnPackDDElParam
FreeDDElParam

Adjusting Calls to Functions

Most of your source code is not affected by differences between the APIs of Windows 3.x and Win32. The underlying definitions in the header files (see WINDOWS.H) automatically adjust data to the correct size. But you may need to revise code if you call functions in any of the following categories:

- Graphics functions
- Functions accessing "extra" window data
- MS-DOS system calls
- Far-pointer functions
- Functions getting list and combo box contents

Graphics Functions

Most of the Windows 3.x functions that must be replaced return packed x- and y-coordinates.

In Windows 3.x, the x- and y-coordinates are 16 bits each and are packed into the 32-bit **(DWORD)** function return value, the largest valid size. In Win32, the coordinates are 32 bits each, totaling 64 bits, and are thus too large to fit into a single return value. Each Windows 3.x function is replaced by a Win32 function with the same name, but with an **Ex** suffix added. The **Ex** functions pass the x- and y-coordinates using an additional parameter instead of a return value. These new functions are supported by both Win32 and Windows 3.x.

Windows 3.x implements these functions with a static library, in order that source code compiled with the new calls will also function in Windows 3.x.

The problematic graphics functions fall into two groups. The first group, functions that set coordinates, are shown below with the Win32 versions:

Windows 3.x function	Portable version of function
MoveTo	**MoveToEx**
OffsetViewportOrg	**OffsetViewportOrgEx**
OffsetWindowOrg	**OffsetWindowOrgEx**
ScaleViewportExt	**ScaleViewportExtEx**
ScaleWindowExt	**ScaleWindowExtEx**
SetBitmapDimension	**SetBitmapDimensionEx**
SetMetaFileBits	**SetMetaFileBitsEx**
SetViewportExt	**SetViewportExtEx**
SetWindowExt	**SetWindowExtEx**
SetWindowOrg	**SetWindowOrgEx**

Each of the functions in the first column returns a value, although application code frequently ignores it. However, even if you do not care about the return value, you must still replace the old function call by the new form. The old functions are not supported under Win32.

Each **Ex** function includes an additional parameter that points to a location to receive data. After the function call, this data provides the same information as the corresponding function's return value. If you do not need this information, you can pass **NULL** to this parameter.

Under Windows 3.x, a call to the **MoveTo** function can be written as follows:

```
MoveTo( hDC, x, y );
```

In the portable version supported by both versions of Windows, the call to **MoveTo** is rewritten as follows. Note that the information returned by **MoveTo** under Windows 3.x is still ignored:

```
MoveToEx( hDC, x, y, NULL );
```

As a general rule, pass **NULL** as the last parameter unless you need to use the x- and y-coordinates returned by the Windows 3.x version. In the latter case, use the procedure outlined in the next few paragraphs, for the **Get** functions.

The second group of functions consists of functions in which the application code normally does use the return value. They are listed in the following table.

Windows 3.x function	Portable version of function
GetAspectRatioFilter	**GetAspectRatioFilterEx**
GetBitmapDimension	**GetBitmapDimensionEx**
GetBrushOrg	**GetBrushOrgEx**

GetCurrentPosition	**GetCurrentPositionEx**
GetTextExtent	**GetTextExtentPoint**
GetTextExtentEx	**GetTextExtentExPoint**
GetViewportExt	**GetViewportExtEx**
GetViewportOrg	**GetViewportOrgEx**
GetWindowExt	**GetWindowExtEx**
GetWindowOrg	**GetWindowOrgEx**

The **GetTextExtent** function uses the **Point** suffix, because there is already a Windows 3.1 extended function **GetTextExtentEx**. Therefore, the **Point** suffix is added to the functions **GetTextExtent** and **GetTextExtentEx**, to name the portable versions for each.

As with the first group of functions, the **Ex** (and **Point**) versions each add an additional parameter: a pointer to a **POINT** or **SIZE** structure to receive x/y coordinates. Because this structure is always the appropriate size for the environment, so you can write portable code by:

- Declaring a local variable of type **POINT** or **SIZE**, as appropriate.
- Passing a pointer to this structure as the last parameter to the function.
- Calling the function. The function responds by filling the structure with the appropriate information.

For example, the Windows 3.x version call to **GetTextExtent** extracts the x- and y-coordinates from a **DWORD** return value (stored in a temporary variable, dwXY):

```
DWORDdwXY;

dwXY = GetTextExtent( hDC, szLabel1, strlen( szFoo ) );
rect.left = 0; rect.bottom = 0;
rect.right = LOWORD(dwXY);
rect.top = HIWORD(dwXY);
InvertRect( hDC, &rect );
```

The portable version passes a pointer to a temporary **SIZE** structure, and then it extracts data from the structure:

```
SIZE sizeRect;

GetTextExtentPoint( hDC, szLabel1, strlen( szLabel1 ), &sizeRect );
rect.left = 0; rect.bottom = 0;
rect.right = sizeRect.cx;
rect.top = sizeRect.cy;
InvertRect( hDC, &rect );
```

Functions That Access the Extra Window Data

The functions described in this topic manipulate the "extra" data area of a window structure. This structure can contain system information as well as user-defined data. You specify the size of this data area by using the **cbClsExtra** member of the **WNDCLASS** structure when you register the window class.

The following Windows 3.x functions get or set 16 bits during each call:

GetClassWord
GetWindowWord
SetClassWord
SetWindowWord

You can use these functions in Windows 3.x to access system information, stored as 16-bit items. But in Win32, each of these system-information items grows to 32 bits. Therefore, in Win32, you would use the following functions which access 32 bits at a time:

GetClassLong
GetWindowLong
SetClassLong
SetWindowLong

Each of these functions take two parameters: a window handle and an offset into the data area. These offsets differ depending on whether you are compiling for Windows 3.x or Win32.

The index values specifying these offsets correspond to each other as follows. Note that neither version is portable.

Windows 3.x	**Win32 (nonportable)**
GCW_CURSOR	GCL_CURSOR
GCW_HBRBACKGROUND	GCL_HBRBACKGROUND
GCW_HICON	GCL_HICON
GWW_HINSTANCE	GWL_HINSTANCE
GWW_HWNDPARENT	GWL_HWNDPARENT
GWW_ID	GWL_ID
GWW_USERDATA	GWL_USERDATA

To create portable code using these offsets, you need to use **#ifdef** statements as shown below. Both the function and the value of the second parameter change:

```
#ifdef WIN32
     hwndParent = (HWND)GetWindowLong(hWnd, GWL_HWNDPARENT);
#else
     hwndParent = (HWND)GetWindowWord(hWnd, GWW_HWNDPARENT);
#endif
```

In the case of **GWW_HWNDPARENT**, you can avoid calls to **GetWindowLong** and **GetWindowWord** altogether, and instead use a single call to a new function, **GetParent**. This function returns a handle of the appropriate size. The following example illustrates a call to **GetParent** that has the same results as the **#ifdef** statements shown in the previous example:

```
hwndParent = GetParent( hWnd );
```

Remember that offsets may change for private data that you store in the Window structure. You should review this code carefully and recalculate offsets for Win32, noting that some data types, such as

handles, increase in size.

Porting MS-DOS System Calls

The **DOS3Call** function in Windows 3.0 must be called from assembly language. It is typically used to perform file I/O. In Win32, assembly language code that calls **DOS3Call** should be replaced by the appropriate Win32 file I/O calls. Other (non-file) INT 21H functions should be replaced, as shown in the following table, with the portable Windows call:

INT 21H subfunction	MS-DOS operation	Win32 API equivalent
0EH	Select Disk	**SetCurrentDirectory**
19H	Get Current Disk	**GetCurrentDirectory**
2AH	Get Date	**GetSystemTime**
2BH	Set Date	**SetSystemTime**
2CH	Get Time	**GetSystemTime**
2DH	Set Time	**SetSystemTime**
36H	Get Disk Free Space	**GetDiskFreeSpace**
39H	Create Directory	**CreateDirectory**
3AH	Remove Directory	**RemoveDirectory**
3BH	Set Current Directory	**SetCurrentDirectory**
3CH	Create Handle	**CreateFile**
3DH	Open Handle	**CreateFile**
3EH	Close Handle	**CloseHandle**
3FH	Read Handle	**ReadFile**
40H	Write Handle	**WriteFile**
41H	Delete File	**DeleteFile**
42H	Move File Pointer	**SetFilePointer**
43H	Get File Attributes	**GetFileAttributes**
43H	Set File Attributes	**SetFileAttributes**
47H	Get Current Directory	**GetCurrentDirectory**
4EH	Find First File	**FindFirstFile**
4FH	Find Next File	**FindNextFile**
56H	Change Directory Entry	**MoveFile**
57H	Get Date/Time of File	**GetFileTime**
57H	Set Date/Time of File	**SetFileTime**
59H	Get Extended Error	**GetLastError**
5AH	Create Unique File	**GetTempFileName**
5BH	Create New File	**CreateFile**
5CH	Lock	**LockFile**
5CH	Unlock	**UnlockFile**
67H	Set Handle Count	**SetHandleCount**

File Operations

Fixed-length buffers for filenames and environment strings may need to be increased in size. Windows NT (one implementation of Win32) supports filenames of up to 256 characters, rather than the 8.3 format supported by MS-DOS. You can make code more portable by allocating longer buffers or by using dynamic memory allocation. If you want to conserve memory under Windows 3.x, use **#ifdef** statements to allocate buffers of the proper length for the environment.

Another area in which you might need to make changes is low-level file I/O. In porting Windows 3.x code, some developers have chosen to change from using the Windows API file I/O functions (such as **_lopen** and **_lread**) to using the C run-time low-level I/O functions (such as **_open** and **_read**). All versions of the Windows API support binary mode only, not text mode, but the C run-time calls use text mode by default. Therefore, when changing from the Windows file I/O to the C run-time versions, open files in binary mode by doing one of the following:

- Link with BINMODE.OBJ, which changes the default mode for all file-open operations.
- Open the individual files with _O_BINARY flag set.
- Use **setmode** to change an open file to _O_BINARY.

Far-Pointer Functions

Windows 3.x provides functions for memory and file manipulation using far pointers, which have the form **_f***xxxx*. In Win32, these functions are replaced by similarly named functions of the form *xxxx*, because there is no need for far pointers in Win32. (The **_f** prefix is dropped from the name.)

The WINDOWSX.H file defines the **_f***xxxx* function names so that in Win32, the **_f***xxxx* function names are equated to the corresponding functions that are still supported. This means that as long as you include WINDOWSX.H, you don't have to rewrite calls to these functions. Some of the definitions are:

```
#define _fmemcpy(x,y,z)    memcpy(x,y,z)
#define _fstrcpy(x,y)        strcpy(x,y)
#define _fstrcmp(x,y)        strcmp(x,y)
#define _fstrcat(x,y)        strcat(x,y)
```

Functions Getting List and Combo Box Contents

The Win32 API contains two new functions, shown in the following table, that provide an improved means of extracting list and combo box contents. In each case, the portable version of the function lets you specify a buffer size for a string that receives the information.

Windows 3.x function	Portable version of the function
DlgDirSelect	**DlgDirSelectEx**
DlgDirSelectComboBox	**DlgDirSelectComboBoxEx**

For example, Windows 3.x code might contain the following function call:

```
DlgDirSelect( hDlg, lpString, nIDListBox );
```

This line of code should be replaced by the following call to **DlgDirSelectEx**:

```
DlgDirSelectEx( hDlg, lpString, sizeof(lpString), nIDListBox );
```

Revising the WinMain Function

You need to revise the **WinMain** function if either of the following conditions is true. Otherwise, the code in this function generally needs no change.

- Your application needs to know when another instance of the application is running, or
- You you need to access the command line

The parameter list for **WinMain** is the same for Win32 and Windows 3.x:

```
APIENTRY WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
```

However, under Win32, there are two differences in the values passed through these parameters:

- Unlike Windows 3.x, *hPrevInstance* always returns NULL.
- The *lpCmdLine* parameter points to a string containing the entire command line, not just the parameters.

The following two sections discuss each of these differences.

Initializing Instances

The *hPrevInstance* parameter always returns **NULL** in Win32. This causes each instance of an application to act as though it were the only instance running. The application must register the window class, and it cannot access data used by other instances, except through standard interprocess communication techniques such as shared memory or DDE. Calls to **GetInstanceData** must be replaced with these techniques.

Source code for Windows 3.x normally tests *hPrevInstance* to see if another instance of the application is already running; if the value is **NULL** (indicating there is no previous instance), the code registers the window class. This code is automatically portable to Win32 and needs no change: the result under Win32 is simply that it always registers the window class, which is correct behavior.

Some applications must know if other instances are running. Sometimes this is because data sharing is required. More frequently, it is because only one instance of the application should run at a time. Examples of this latter case include Control Panel and Task Manager.

Applications cannot use *hPrevInstance* to test for previous instances under Win32. An alternative method must be used, such as creating a unique named pipe, creating/testing for a named semaphore, broadcasting a unique message, or calling **FindWindow**.

Accessing the Command Line Through LpCmdLine

In Windows 3.x, the *lpCmdLine* parameter points to a string containing the command line starting with the first parameter. The name of the application is not included. In Win32, *lpCmdLine* points to a string containing the entire command line, including the application itself.

Special Considerations for Advanced Applications

If you have applied the guidelines and procedures described in the preceding topics, you may well have been able to revise your entire application so that it is portable. However, problems specific to applications that use advanced calls or C coding tricks require additional revision. If your application is fairly complex, you should scan through the next two topics to make sure that you don't need additional changes.

Revising Advanced Function Calls

Applications may need further revision if they use calls dealing with any of the following: accessing .INI files, setting focus and active window, capturing the mouse, and sharing graphical objects.

Profile Strings and .INI Files

Although Windows NT and Win32s are both examples of Win32, Windows NT presents some additional features not present in Win32s.

Windows 3.x applications can access .INI files directly. In Windows NT, however, such code doesn't work because the information in .INI files is replaced by a registration database. This database offers a number of advantages, including security controls that prevent an application from corrupting system information, error logging, remote software updating, and remote administration of workstation software.

You can write portable code by using the profile API supported by Windows 3.x and all versions of Win32, including Windows NT. Call the **GetProfileString** and **WriteProfileString** functions instead of accessing .INI files directly. These functions use whichever underlying mechanism (.INI file or registration database) is supported by the environment you are compiling for.

Focus, Mouse Capture, and Localized Input

The Windows NT environment differs from Windows 3.x in that each thread of execution has its own message queue. This change affects window focus and mouse capture.

Window Focus

In Windows NT, each thread of execution can set or get the focus only to windows created by the current thread. This behavior prevents applications from interfering with each other. One application's delay in responding cannot cause other applications to suspend their response to user actions, as often happens in Windows 3.x.

Consequently, the following functions work differently under Windows NT:

GetActiveWindow
GetCapture
GetFocus
ReleaseCapture
SetActiveWindow
SetCapture
SetFocus

Get functions can now return **NULL**, which could not happen in Windows 3.x. Therefore, it's important to test the return value of **GetFocus** before using it. Instead of returning the window handle of another thread, the function returns **NULL**. For example, you call **GetFocus** and another thread has the focus. Note that it's possible for a call to **GetFocus** to return **NULL** even though an earlier call to **SetFocus** successfully set the focus. Simliar considerations apply to **GetCapture** and **GetActiveWindow**.

The **Set** functions can only specify a window created by the current thread. If you attempt to pass a window handle created by another thread, the call to the **Set** function fails.

Mouse Capture

Mouse capture is also affected by the Windows NT localized input queues. If the mouse is captured on mouse down, the window capturing the mouse receives mouse input until the mouse button is released, as in Windows 3.x. But if the mouse is captured while the mouse button is up, the window receives mouse input only as long as the mouse is over that window or another window created by the same thread.

Shared Graphical Objects

Win32 applications run in separate address spaces. Graphical objects are specific to the application and cannot be manipulated by other processes as in Windows 3.x. A handle to a bitmap passed to another process cannot be used because the original process retains ownership.

Each process should create its own pens and brushes. A cooperative process may access the bitmap data in shared memory (by way of standard interprocess communications) and create its own copy of the bitmap. Alterations to the bitmap must be communicated between the cooperative processes by way of interprocess communication.

Solving Problems Due to C Coding Techniques

Some portability problems can be caused by C coding techniques that do not translate successfully to other memory models and processors. You can avoid these problems if your code doesn't attempt to exploit the fact you know about segmented memory. Standard pointer and memory management techniques will be correctly resolved by the header files.

The coding techniques most likely to cause porting problems are described in the following sections. For a more thorough general treatment of this subject, read Writing Portable C Programs.

Memory and Pointers

To be portable, source code must avoid any techniques that rely on the 16-bit *segment*:*offset* address structure, because all pointers are 32 bits in size under Win32 and use flat rather than segmented memory.

This difference in pointer structure is usually not a problem unless the code uses **HIWORD**, **LOWORD**, or similar macros to manipulate portions of the pointer.

For example, in Windows 3.x, memory is allocated to align on a segment boundary, which makes memory allocation functions return a pointer with an offset of `0x0000`. The following code exploits this fact to run successfully under Windows 3.x:

```
ptr2 = ptr1 = malloc();                 // ptr2 = xxxx:0000
LOWORD( ptr2 ) = index * elementsize;   // Place offset of array element
                                        //   into ptr2 low word
```

Such code does not work properly under Win32. But standard pointer constructs, such as the following, always result in portable code:

```
ptr1 = malloc();            // Set ptr1 to start of memory block
ptr2 = ptr1[i];             // Place offset of array element
```

Here are some other guidelines for dealing with pointers:

- All pointers, including those that access the local heap, are 32 bits under Win32.
- Addresses never wrap, as they can with the low word in segmented addressing; for example, in Windows 3.x, an address can wrap from 1000:FFFF to 1000:0000.
- Structures that hold near pointers in Windows 3.x must be revised because all pointers are 32 bits in Win 32. This may affect code that uses constants to access structure members, and it may also affect alignment.

Structure Alignment

Applications should generally align structure members at addresses that are "natural" for the data type and the processor involved. For example, a 4-byte data member should have an address that is a multiple of four.

This principle is especially important when you write code for porting to multiple processors. A misaligned 4-byte data member, which is on an address that is not a multiple of four, causes a performance penalty with an 80386 processor and a hardware exception with a RISC processor. In the latter cases, although the exception is handled by the system, the performance penalty is significantly greater.

Alignment problems can be avoided by setting compiler options or adjusting your structure definitions to meet the alignment requirements. The guidelines following ensure proper alignment for processors targeted by Win32:

Type	**Alignment**
char	Align on byte boundaries
short (16-bit)	Align on even byte boundaries
int and **long** (32-bit)	Align on 32-bit boundaries
float	Align on 32-bit boundaries
double	Align on 64-bit boundaries
structures	Align on 32-bit boundaries

Ranges and Promotions

Occurrences of **int**, **unsigned**, and **unsigned int** indicate potential portability problems because size and range are not constant. Data that would not exceed its range in Win32 could exceed range in Windows 3.x. Sign extension also works differently, so exercise caution in performing bitwise manipulation of this data.

Source code that relies on wrapping often presents portability problems, and should be avoided. For example, a loop should not rely on an **unsigned int** variable wrapping at 65535 (the maximum value in Windows 3.x) back down to 0.

Dynamic-Link Libraries

Development of dynamic-link libraries (DLLs) is different under Win32 compared to Windows 3.x. There are differences in the initialization and termination routines: how often they are called, what is passed, and how much code must be written in assembly language:

- In Win32 DLLs, one function handles both initialization and termination. In Windows 3.x, the initialization function must be provided, and the termination function, if present, must be named WEP.
- The Win32 initialization function is called every time a new process or an additional thread accesses the DLL for the first time, and when a process or thread detaches. In Windows 3.x, initialization and termination functions are called only once during the lifetime of the DLL.
- In Win32, the entire DLL can be written in C (which is recommended as an aid to porting to other processors). In Windows 3.x, the start-up code is linked to an object file written in assembly language. Microsoft C/C++ links in an object file, LIBENTRY.OBJ, that accesses information in information in registers and calls **LibMain** in the C code. In Win32, you don't need this file.

In Win32, the DLL initialization function is the same as the termination function. By convention this function is named **DllMain**. A **DWORD** (32-bit) parameter, *dwReason*, notifies the function whether initialization or termination is taking place, and whether a process or a thread is involved. When a process first accesses a DLL, the initialization function is called with *process-attach* notification: it is assumed that the process has one thread to being with. When there is an access by an additional thread of that process, the function is called with *thread-attach* notification. However, in Win32s there is notification only for process attach and detach.

The Win32 DLL initialization function should return 1 to indicate success. Returning NULL indicates failure.

Windows 3.x DLL initialization functions are passed the following information:

- The DLL's instance handle
- The DLL's data segment (DS)
- The heap size specified in the DLL's .DEF file
- The command line

Win32 DLL initialization functions are passed the following information:

- The *hModule* parameter, a module handle
- The *dwReason* parameter, an enumerated type which indicates which of four reasons the LibMain procedure is being called: process attach, thread attach, thread detach, or process detach.
- The *lpReserved* parameter, which is unused.

The definition Win32 initialization function should look like the following code:

```
BOOL APIENTRY DllMain( HANDLE hModule,
                                    DWORD dwReason,
                                    LPVOID lpReserved )
{
     switch( dwReason) {
     case DLL_PROCESS_ATTACH:
     ...
     case DLL_THREAD_ATTACH:
     ...
     case DLL_THREAD_DETACH:
     ...
     case DLL_PROCESS_DETACH:
```

```
        . . .
        }
}
```

The Win32 module handle has the same purpose as the Windows 3.x instance handle. Otherwise, Win32 DLL initialization functions do not include the parameters for Windows 3.x initialization, as described in the following table.

Parameter	Comment
DLL data segment	Not needed in Win32; memory model is flat, not segmented.
Size of DLL's local heap	All calls to local memory management functions operate on the default heap
Pointer to command line	The command line can be obtained through a call to the **GetCommandLine** function.

Windows 3.x assembly code is often limited to the LIBENTRY.ASM module, which simply accesses information in registers and passes it to a **LibMain** routine written in C. **DllMain** will be called directly in Win32, and **LibMain** will be called indirectly in Windows 3.x through the LIBENTRY code. If more of your DLL is written in assembly language, it is a good idea to rewrite all of it in C, to make it more portable.

Summary of Function and Message Differences

The following table provides a list of function calls and messages that required implementation changes for Win32.

Function/Message	Support	Comments
AccessResource	Dropped	No Win32 equivalent
AddFontResource	Enhanced	Must use string, not handle, for filename
AllocDSToCSAlias	Dropped	No Win32 equivalent
AllocResource	Dropped	No Win32 equivalent
AllocSelector	Dropped	No Win32 equivalent
ChangeSelector	Dropped	No Win32 equivalent
CloseComm	Dropped	Replaced by **CloseHandle**
CloseSound	Dropped	Replaced by multimedia sound support
CountVoiceNotes	Dropped	Replaced by multimedia sound support
DefineHandleTable	Dropped	No Win32 equivalent
DeviceMode	Dropped	Replaced by portable **DocumentProperties**
DlgDirSelect	Dropped	Replaced by portable **DlgDirSelectEx**
DlgDirSelectComboBox	Dropped	Replaced by portable **DlgDirSelectComboBoxEx**
DOS3Call	Dropped	Replaced by named, portable Win32 function
ExtDeviceMode	Dropped	Replaced by portable **DocumentProperties**
FlushComm	Dropped	Replaced by **PurgeComm**
FreeSelector	Dropped	No Win32 equivalent
GetAspectRatioFilter	Dropped	Replaced by portable **GetAspectRatioFilterEx**
GetBitmapDimension	Dropped	Replaced by portable **GetBitmapDimensionEx**
GetBrushOrg	Dropped	Replaced by portable **GetBrushOrgEx**
GetClassWord	Enhanced	Use **GetClassLong** for values that grow to 32-bits in Win32
GetCodeHandle	Dropped	No Win32 equivalent
GetCodeInfo	Dropped	No Win32 equivalent
GetCommError	Dropped	Replaced by **GetCommState**
GetCurrentPDB	Dropped	No Win32 equivalent
GetCurrentPosition	Dropped	Replaced by portable **GetCurrentPositionEx**
GetEnvironment	Dropped	No Win32 equivalent
GetFreeSpace	Dropped	Replaced by **GlobalMemoryStatus**

GetFreeSystemResources	Dropped	Replaced by **GlobalMemoryStatus**
GetInstanceData	Dropped	No equivalent; use alternative supported IPC mechanism.
GetKBCodePage	Dropped	No Win32 function equivalent
GetMetaFileBits	Dropped	Replaced by portable **GetMetaFileBitsEx**
GetModuleUsage	Enhanced	Always returns 1 on Win32
GetTempDrive	Dropped	Replaced by portable **GetTempPath**
GetTextExtent	Dropped	Replaced by portable **GetTextExtentPoint**
GetTextExtentEx	Dropped	Replaced by portable **GetTextExtentExPoint**
GetThresholdEvent	Dropped	Replaced by multimedia sound support
GetThresholdStatus	Dropped	Replaced by multimedia sound support
GetViewportExt	Dropped	Replaced by portable **GetViewportExtEx**
GetViewportOrg	Dropped	Replaced by portable **GetViewportOrgEx**
GetWindowExt	Dropped	Replaced by portable **GetWindowExtEx**
GetWindowOrg	Dropped	Replaced by portable **GetWindowOrgEx**
GetWindowWord	Enhanced	Use **GetWindowLong** for values that grow to 32-bits on Win32
GlobalCompact	Dropped	No Win32 equivalent
GlobalDosAlloc	Dropped	No Win32 equivalent
GlobalDosFree	Dropped	No Win32 equivalent
GlobalFix	Dropped	No Win32 equivalent
GlobalLRUNewest	Dropped	No Win32 equivalent
GlobalLRUOldest	Dropped	No Win32 equivalent
GlobalNotify	Dropped	No Win32 equivalent
GlobalPageLock	Dropped	No Win32 equivalent
GlobalPageUnlock	Dropped	No Win32 equivalent
GlobalUnfix	Dropped	No Win32 equivalent
GlobalUnwire	Dropped	No Win32 equivalent
GlobalWire	Dropped	No Win32 equivalent
LimitEmsPages	Dropped	No Win32 equivalent
LocalCompact	Dropped	No Win32 equivalent
LocalInit	Dropped	No Win32 equivalent
LocalNotify	Dropped	No Win32 equivalent
LocalShrink	Dropped	No Win32 equivalent
LockSegment	Dropped	No Win32 equivalent

MoveTo	Dropped	Replaced by portable **MoveToEx**
NetBIOSCall	Dropped	Replaced by named, portable Win32 function
OffsetViewportOrg	Dropped	Replaced by portable **OffsetViewportOrgEx**
OffsetWindowOrg	Dropped	Replaced by portable **OffsetWindowOrgEx**
OpenComm	Dropped	Replaced by **OpenFile**
OpenSound	Dropped	Replaced by multimedia sound support
ProfClear	Dropped	Replaced by Win32 profile-string function
ProfFinish	Dropped	Replaced by Win32 profile-string function
ProfFlush	Dropped	Replaced by Win32 profile-string function
ProfInsChk	Dropped	Replaced by Win32 profile-string function
ProfSampRate	Dropped	Replaced by Win32 profile-string function
ProfSetup	Dropped	Replaced by Win32 profile-string function
ProfStart	Dropped	Replaced by Win32 profile-string function
ProfStop	Dropped	Replaced by Win32 profile-string function
ReadComm	Dropped	Replaced by **ReadFile**
RemoveFontResource	Enhanced	Must use string, not handle, for filename
ScaleViewportExt	Dropped	Replaced by portable **ScaleViewportExtEx**
ScaleWindowExt	Dropped	Replaced by portable **ScaleWindowExtEx**
SetBitmapDimension	Dropped	Replaced by portable **SetBitmapDimensionEx**
SetClassWord	Enhanced	Use **SetClassLong** for values that grow to 32-bits on Win32
SetCommEventMask	Dropped	Replaced by **SetCommMask**
SetEnvironment	Dropped	No Win32 equivalent
SetMetaFileBits	Dropped	Replaced by portable **SetMetaFileBitsEx**
SetResourceHandler	Dropped	No Win32 equivalent
SetSoundNoise	Dropped	Replaced by multimedia sound support
SetSwapAreaSize	Dropped	No Win32 equivalent
SetViewportExt	Dropped	Replaced by portable **SetViewportExtEx**
SetViewportOrg	Dropped	Replaced by portable

		SetViewportOrgEx
SetVoiceAccent	Dropped	Replaced by multimedia sound support
SetVoiceEnvelope	Dropped	Replaced by multimedia sound support
SetVoiceNote	Dropped	Replaced by multimedia sound support
SetVoiceQueueSize	Dropped	Replaced by multimedia sound support
SetVoiceSound	Dropped	Replaced by multimedia sound support
SetVoiceThreshold	Dropped	Replaced by multimedia sound support
SetWindowExt	Dropped	Replaced by portable **SetWindowExtEx**
SetWindowOrg	Dropped	Replaced by portable **SetWindowOrgEx**
SetWindowWord	Enhanced	Use **SetWindowLong** for values that grow to 32-bits on Win32
StartSound	Dropped	Replaced by multimedia sound support
StopSound	Dropped	Replaced by multimedia sound support
SwitchStackBack	Dropped	No Win32 equivalent
SwitchStackTo	Dropped	No Win32 equivalent
SyncAllVoices	Dropped	Replaced by multimedia sound support
UngetCommChar	Dropped	No Win32 equivalent
UnlockSegment	Dropped	No Win32 equivalent
ValidateCodeSegments	Dropped	No Win32 equivalent
ValidateFreeSpaces	Dropped	No Win32 equivalent
WaitSoundState	Dropped	Replaced by multimedia sound support
WriteComm	Dropped	Replaced by **WriteFile**
EM_GETSEL	Enhanced	*wParam/lParam* packing changed
EM_LINESCROLL	Enhanced	*wParam/lParam* packing changed
EM_SETSEL	Enhanced	*wParam/lParam* packing changed
WM_ACTIVATE	Enhanced	*wParam/lParam* packing changed
WM_CHANGECBCHAIN	Enhanced	*wParam/lParam* packing changed
WM_CHARTOITEM	Enhanced	*wParam/lParam* packing changed
WM_COMMAND	Enhanced	*wParam/lParam* packing changed

WM_CTLCOLOR	Dropped	Replaced by WM_CTLCOLOR<*type*> messages
WM_DDE_ACK	Enhanced	*wParam/lParam* packing changed
WM_DDE_ADVISE	Enhanced	*wParam/lParam* packing changed
WM_DDE_DATA	Enhanced	*wParam/lParam* packing changed
WM_DDE_EXECUTE	Enhanced	*wParam/lParam* packing changed
WM_DDE_POKE	Enhanced	*wParam/lParam* packing changed
WM_HSCROLL	Enhanced	*wParam/lParam* packing changed
WM_MDIACTIVATE	Enhanced	*wParam/lParam* packing changed
WM_MDISETMENU	Enhanced	*wParam/lParam* packing changed
WM_MENUCHAR	Enhanced	*wParam/lParam* packing changed
WM_MENUSELECT	Enhanced	*wParam/lParam* packing changed
WM_PARENTNOTIFY	Enhanced	*wParam/lParam* packing changed
WM_VKEYTOITEM	Enhanced	*wParam/lParam* packing changed
WM_VSCROLL	Enhanced	*wParam/lParam* packing changed
DCB	Enhanced	Changes to bitfields and additional structure members
(WORD)	16-bit	Check if incorrect cast of 32-bit value
GCW_HCURSOR	Dropped	Replaced by GCL_HCURSOR
GCW_HBRBACKGROUND	Dropped	Replaced by GCL_HBRBACKGROUND
GCW_HICON	Dropped	Replaced by GCL_HICON
GWW_HINSTANCE	Dropped	Replaced by GWL_HINSTANCE
GWW_HWNDPARENT	Dropped	Replaced by GWL_HWNDPARENT
GWW_ID	Dropped	Replaced by GWL_ID
GWW_USERDATA	Dropped	Replaced by GWL_USERDATA
READ	Dropped	Replaced by OF_READ
WRITE	Dropped	Replaced by OF_WRITE
READ_WRITE	Dropped	Replaced by OF_READ_WRITE
HIWORD	16-bit	Check if **HIWORD** target is 16- or 32-bit
LOWORD	16-bit	Check if **LOWORD** target is 16-

		or 32-bit
MAKEPOINT	Dropped	Replaced by **LONG2POINT**

Handling Messages with Portable Macros

Instead of taking the case-by-case approach, you can use message-cracking macros to write message handlers similar to those you'd write when using Microsoft Foundation Classes. These message handlers use the same parameter list regardless of operating system, thereby solving message-packing issues. This guide describes these and other macros defined in WINDOWSX.H (or WINDOWSX.H16 in the case of 16-bit applications).

This guide includes the following topics:

Using Message Crackers

Message crackers are a set of macros that extract useful information from the *wParam* and *lParam* parameters of a message and hide the details of how information is packed.

Using message crackers initially requires you to revise some of your code. They also have a minor impact on performance by involving an additional function call. However, they offer the following major advantages:

- Portability. Message crackers free you from packing issues and guarantee proper extraction of information, regardless of which environment you're compiling for.
- Readability. With message crackers, you can understand source code because message parameters are translated into data with meaningful names.
- Ease of use. In addition to decoding *wParam* and *lParam*, message crackers place message-handling code in separate functions. Instead of a long **switch** statement, you have a separate handler for each message.

Overview of Message Crackers

You use message crackers in your code by writing a separate message handler function for each message. Then you use a macro to call each of those functions from within your window procedure.

Use of message crackers for all messages is recommended, but you can optionally combine code that uses message crackers for some messages with code that responds to other messages directly.

Note To use message crackers, make sure you include the file WINDOWSX.H (or WINDOWSX.H16, in the case of 16-bit applications).

Suppose you have a message, WM_THIS. The code to handle this message would look something like this:

```
LONG WINAPI My_WndProc( HWND hwnd, UINT msg, UINT wParam, LONG lParam )
.
.
.
switch( msg ) {
     case WM_THIS:
           // Place code to handle message here
```

To use message crackers, write a message handler, and then call it from the **switch** statement. Suppose that there are two pieces of information contained in the WM_THIS message: *thisHdc* and *thisData*. Message crackers unpack this information from *wParam* and *lParam*, and pass it as parameters to your message handler, MyWnd_OnThis:

```
switch ( msg) {
     case WM_THIS:
           return HANDLE_WM_THIS ( hwnd, wParam, lParam, MyWnd_OnThis );
...
}


LRESULT MyWnd_OnThis ( HWND hwnd, HDC thisHdc, WORD thisData )
{
     // Place code to handle message here
}
```

The parameters to MyWnd_OnThis (after *hwnd*, which is always the first parameter) consist of information directly usable by your code: *thisHdc* and *thisData*. The macro **HANDLE_WM_THIS** translates *wParam* and *lParam* into *thisHdc* and *thisData* as it makes the function call.

The following general steps summarize how to use message crackers:

1. Declare a prototype for each message-handling function.
2. In the windows procedure, call the message handler. Use either a message decoder (such as **HANDLE_WM_CREATE**) or the **HANDLE_MSG** macro.
3. Write the message handler. Use a message forwarder such as **FORWARD_WM_CREATE** to call the default message procedure.

Declaring Message-Handler Prototypes

To use message crackers, first declare a prototype for the message handling function ("message handler" for short). Although you can give your message handlers any name you want, a recommended convention is:

*WndClass*_**On***Msg*

in which *WndClass* is the name of the window class, and *Msg* is the name of the message in mixed case, with the "WM" dropped. For example, the following code contains prototypes for functions handling WM_CREATE, WM_PAINT, and WM_MOUSEMOVE:

```
BOOL MyWnd_OnCreate( HWND hwnd, CREATESTRUCT FAR* lpCreateStruct );
void MyWnd_OnPaint( HWND hwnd );
void MyWnd_OnMouseMove( HWND hwnd, int x, int y, UINT keyFlags );
```

The first parameter to each function is always *hwnd*, which is a handle to the window that received the message. The rest of the parameters vary; each message handler has its own customized parameter list. To declare the appropriate parameters for a message, see the corresponding definitions in WINDOWSX.H.

Calling the Message Handler

In your window procedure, you call a message handler by using a message-decoder macro such as **HANDLE_WM_CREATE** or **HANDLE_WM_PAINT**. The general form for using these macros is:

case *msg*:
    return HANDLE_*msg* **(hwnd, wParam, lParam,** *handler* **);**

You should always return the value of the macro, even if no return value is expected and the corresponding message handler has **void** return type.

For example, you could place the following macros in your code:

```
switch( msg ) {
     case WM_CREATE:
           return HANDLE_WM_CREATE( hwnd, wParam, lParam, MyWnd_OnCreate );
     case WM_PAINT:
           return HANDLE_WM_PAINT( hwnd, wParam, lParam, MyWnd_OnPaint );
     case WM_MOUSEMOVE:
           return HANDLE_WM_MOUSEMOVE( hwnd, wParam, lParam,
                                       MyWnd_OnMouseMove );
     .
     .
     .
```

Alternatively, you can use the generic **HANDLE_MSG** macro, which generates the same code as the previous example, but saves space:

```
switch( msg ) {
     HANDLE_MSG( hwnd, WM_CREATE, MyWnd_OnCreate );
     HANDLE_MSG( hwnd, WM_PAINT, MyWnd_OnPaint );
     HANDLE_MSG( hwnd, WM_MOUSEMOVE, MyWnd_OnMouseMove );
     .
     .
     .
```

HANDLE_MSG assumes that you use the names *wParam* and *lParam* in the window procedure parameter list. You cannot use this macro if you have given these parameters other names.

Writing the Message Handler

In the message-handling function, you respond to the message using parameters that have been translated from *wParam* and *lParam* and passed to the function. In the following example, *lpCreateStruct* is an example of a parameter translated from *wParam* and *lParam*:

```
BOOL MyCls_OnCreate(HWND hwnd, CREATESTRUCT FAR* lpCreateStruct)
{
     // Place message-handling code here

    return FORWARD_WM_CREATE(hwnd, lpCreateStruct, DefWindowProc);
}
```

Message-handling code often finishes by calling **DefWindowProc** or some other default message procedure. You make this function call by using a "message forwarder," which uses the following form:

return FORWARD_*msg***(** *parmlist***,** *defaultMsgProc* **);**

The *parmlist* is the same list of parameters in the message handler, and *defaultMsgProc* is the default message procedure, typically **DefWindowProc**. The message forwarder repacks the information in the parameter list into the appropriate *wParam*/*lParam* format (depending on target environment) and forwards the message to the default message procedure.

Putting It Together: An Example

The following example demonstrates the use of several message handlers in a window procedure, showing where the various prototypes and macros fit into the code.

The header file, MYAPP.H, consists of function prototypes, including prototypes for the message handlers. Note how each message handler has its own parameter list, which is customized to best represent the information packed in the corresponding message:

```
// MYAPP.H

// Window procedure prototype

LONG WINAPI MyWnd_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam);

// Default message handler

#define MyWnd_DefProc    DefWindowProc

// MyWnd class message handler functions, declared in a .h file:
//
void MyWnd_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags);
void MyWnd_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT
keyFlags);
void MyWnd_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags);
```

The rest of the code in this example is in MYAPP.C, which contains the window procedure and the individual message handlers. With message crackers, the function of the window procedure is principally to route each message to the appropriate handler.

Both the WM_LBUTTONDOWN and WM_LBUTTONDBLCLK messages map to the MyWnd_OnLButtonDown procedure. This mapping is one of the special cases of message handling described in Handling Special Cases of Messages.

```
// MYAPP.C
--------------------------------------------------------------------

// MyWnd window procedure implementation.
//
LONG WINAPI MyWnd_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM Param)
{
    switch (msg)
    {
        HANDLE_MSG(hwnd, WM_MOUSEMOVE,MyWnd_OnMouseMove);
        HANDLE_MSG(hwnd, WM_LBUTTONDOWN, MyWnd_OnLButtonDown);
        HANDLE_MSG(hwnd, WM_LBUTTONDBLCLK, MyWnd_OnLButtonDown);
        HANDLE_MSG(hwnd, WM_LBUTTONUP, MyWnd_OnLButtonUp);
    default:
        return MyWnd_DefProc(hwnd, msg, wParam, lParam);
    }
}

// Message handler function implementations:
//
void MyWnd_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags)
```

```
{
    .
     .
     .
     return FORWARD_WM_MOUSEMOVE( hwnd, x, y, keyFlags, MyWnd_DefProc);
}

void MyWnd_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT
keyFlags)
{
    .
     .
     .
     return FORWARD_WM_LBUTTONDOWN( hwnd, fDoubleClick, x, y, keyFlags,
MyWnd_DefProc);
}

void MyWnd_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags)
{
    .
     .
     .
     return FORWARD_WM_LBUTTONUP( hwnd, x, y, keyFlags, MyWnd_DefProc);
}
```

Note that the symbol MyWnd_DefProc is defined to represent **DefWindowProc**. The purpose of this definition is to make code more reusable. This approach assumes you have a similar definition in each application. For example, in an MDI child control procedure, you would have this definition:

```
#define MyWnd_DefProc    DefMDIChildProc
```

If you then copied your message handler to the MDI procedure, you would only need to change the prefix in MyWnd_DefProc to make the code you copied work correctly. Conversely, if your code used the explicit call to **DefWindowProc**, it could create a bug that would be difficult to track down when copied to the MDI code.

Handling Special Cases of Messages

As a general rule, there is one set of message crackers for each message: a message decoder and a message forwarder. Another rule is that each message handler you write should return the same value that your code would normally return for that message. The following messages present exceptions to these rules:

Message handler	Comment
OnCreate, OnNCCreate	**BOOL** return type: return TRUE if there are no errors. If FALSE is returned, a window will not be created.
OnKey	Handles both key up and key down messages. The extra parameter *fDown* indicates whether the key is down or up.
OnLButtonDown, OnRButtonDown	Handles both click (button down) and double-click messages. The extra parameter *fDoubleClick* indicates whether the message received is a double-click message.
OnChar	This handler is passed only by character, and not the virtual key or key flags information.

Writing Message Crackers for User-Defined Messages

You can use message crackers with window messages that you define, but you must write your own macros. The easiest way to do this is to copy and modify existing macros from WINDOWSX.H.

To understand how to write these macros, consider some of the message crackers defined in WINDOWSX.H:

```
/* BOOL Cls_OnCreate(HWND hwnd, CREATESTRUCT FAR* lpCreateStruct) */

#define HANDLE_WM_CREATE(hwnd, wParam, lParam, fn) \
    ((fn)(hwnd, (CREATESTRUCT FAR*)lParam) ? 0L : (LRESULT)-1L)

#define FORWARD_WM_CREATE(hwnd, lpCreateStruct, fn) \
    (BOOL)(DWORD)(fn)(hwnd, WM_CREATE, 0, (LPARAM)lpCreateStruct)
```

The message decoder (**HANDLE_***msg*) should be defined as a function call, (fn), followed by *hwnd* and other parameters derived from *wParam* and *lParam*. The message forwarder (**FORWARD_***msg*) performs the reverse operation on the parameters, putting information back together to restore *wParam* and *lParam* before making the function call (fn). Each of these macros must cast the return value so that the correct type is returned.

When calling the message crackers you write, be careful about variable message values. If your message value is a constant (such as WM_USER+100), you can use HANDLE_MSG with the message in a **switch** statement. However, if the message is registered with **RegisterWindowMessage**, it assigns a number at run time. In this situation, you can't use HANDLE_MSG, because variables cannot be used as **case** values. You must handle the message separately, in an **if** statement:

```
// In MyWnd class initialization code:
//
UINT WM_NEWMESSAGE= 0;

WM_NEWMESSAGE= RegisterWindowMessage("WM_NEWMESSAGE");
.
.
.
// In MyWnd_WndProc(): window procedure:
//
LONG WINAPI MyWnd_WndProc(HWND hwnd, WORD msg, WPARAM wParam,
                          LPARAM lParam)
{
    if (msg == WM_NEWMESSAGE)
        HANDLE_WM_NEWMESSAGE(hwnd, wParam, lParam, MyWnd_OnNewMessage);

    switch (msg)
    {
        HANDLE_MSG(hwnd, WM_MOUSEMOVE, MyWnd_OnMouseMove);
     .
     .
     .
}
```

Adapting Message Crackers to Special Cases

Message crackers can generally be used with all types of application code. However, certain situations require modifications in coding style.

Dialog Procedures, Window Subclassing, and Window Instance Data show how to adapt message-cracker coding techniques for dialog procedures, window subclassing, and window instance data.

Dialog Procedures

Dialog procedures return a **BOOL** value to indicate whether the message was processed. (Window procedures, in contrast, return a **LONG** value rather than a **BOOL**.) Therefore, to adapt a message cracker to dialog-procedure code, you must call the message handler and cast the value to **BOOL**.

Because you have to insert the **(BOOL)** cast, you can't use HANDLE_MSG. You must invoke the message-decoder macro explicitly. Here's an example that shows how you'd use message crackers in a dialog procedure:

```
BOOL MyDlg_OnInitDialog(HWND hwndDlg, HWND hwndFocus, LPARAM lParam);
void MyDlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);

BOOL WINAPI MyDlg_DlgProc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    switch (msg)
    {
    //
    // Since HANDLE_WM_INITDIALOG returns an LRESULT,
    // we must cast it to a BOOL before returning.
    //
    case WM_INITDIALOG:
        return (BOOL)HANDLE_WM_INITDIALOG(hwndDlg, wParam, lParam,
MyDlg_OnInitDialog);

    case WM_COMMAND:
        HANDLE_WM_COMMAND(hwndDlg, wParam, lParam, MyDlg_OnCommand);
        return TRUE;
        break;

    default:
        return FALSE;
    }
}
```

Window Subclassing

When you use message crackers with a subclassed window procedure, the strategy described earlier for using message forwarders does not work. Recall that this strategy involves the following macro call:

return FORWARD_*msg***(** *parmlist***,** *defaultMsgProc* **);**

This use of a message forwarder (**FORWARD_***msg*) calls *defaultMsgProc* directly. But in a subclassed window procedure, you must call the window procedure of the superclass by using the API function **CallWindowProc**. The problem is that **FORWARD_***msg* calls the *defaultMsgProc* with four parameters, but **CallWindowProc** needs five parameters.

The solution is to write an intermediate procedure. For example, the intermediate procedure could be named test_DefProc:

```
        FORWARD_WM_CHAR(hwnd, ch, cRepeat, test_DefProc);
```

The test_DefProc function calls **CallWindowProc** and prepends the address of the superclass function (in this case, test_lpfnwpDefProc) to the parameter list:

```
LONG test_DefProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
     return CallWindowProc(test_lpfnwpDefProc, hwnd, msg, wParam, lParam);
}
```

You need to write just one such procedure for each subclassed window in your application. Each time you use a message forwarder, you give this intermediate procedure as the function address instead of **DefWindowProc**. The following example code shows the complete context:

```
// Global variable that holds the previous window proc address of
// the subclassed window:
//
WNDPROC test_lpfnwpDefProc = NULL;

// Code fragment to subclass a window and store previous wndproc value:
//
void Subclasstest(HWND hwndtest)
{
    extern HINSTANCE g_hinsttest;    // Global application instance handle

    // SubclassWindow() is a macro API that calls SetWindowLong()
    // as appropriate to change the window proc of hwndtest.
    //
    test_lpfnwpDefProc = SubclassWindow(hwndtest,
            (WNDPROC)MakeProcInstance( (FARPROC)test_WndProc, g_hinsttest));
     .
     .
```

```
     .}

// Default message handler function
//
// This function invokes the superclasses' window procedure.  It
// must be declared with the same signature as any window proc,
// so it can be used with the FORWARD_WM_* macros.
//
LONG test_DefProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    return CallWindowProc(test_lpfnwpDefProc, hwnd, msg, wParam, lParam);
}
// test window procedure.  Everything here is the same as in the
// normal non-subclassed case: the differences are encapsulated in
// test_DefProc.
//
LONG WINAPI test_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        HANDLE_MSG(hwnd, WM_CHAR, test_OnChar);
        .
           .
           .
    default:
        //
        // Be sure to call test_DefProc(), NOT DefWindowProc()!
        //
        return test_DefProc(hwnd, msg, wParam, lParam);
    }
}

// Message handlers
//
void test_OnChar(HWND hwnd, UINT ch, int cRepeat)
{
    if (ch == testvalue)
    {
        // handle it here
    }
    else
    {
        // Forward the message on to test_DefProc
        //
        FORWARD_WM_CHAR(hwnd, ch, cRepeat, test_DefProc);
    }
}
```

Window Instance Data

It is common for a window to have user-declared state variables (or "instance data") kept in a separate data structure allocated by the application. You associate this data structure with its corresponding window by storing a pointer to the structure in a specially named window property or in a window word (allocated by setting the cbWndExtra field of the **WNDCLASS** structure when the class is registered).

Message crackers can be adapted to work with this use of instance data. Place the *hwnd* of the window in the first member of the structure. Then, in the message decoders (**HANDLE_***msg* macros), pass the address of the structure instead of the *hwnd*. The message handler now gets a pointer to the structure instead of the *hwnd*, but it can access the *hwnd* through indirection. You may need to rewrite some of the message handler to make it use indirection to access the window handle.

The following example illustrates this technique:

```
// Window instance data structure.  Must include window handle field.
//
typedef struct _test
{
    HWND hwnd;
    int otherStuff;
} test;

// "test" window class was registered with cbWndExtra = sizeof(test*), so we
// can use a window word to store back pointer.  Window properties can also
// be used.
//
// These macros get and set the pointer to the instance data corresponding
to the
// window.  Use GetWindowWord or GetWindowLong as appropriate based on the
default
// size of data pointers.
//
#ifdef WIN32
#define test_GetPtr(hwnd)        (test*)GetWindowLong((hwnd), 0)
#define test_SetPtr(hwnd, ptest)  (test*)SetWindowLong((hwnd), 0, (LONG)
(ptest))
#else
#define test_GetPtr(hwnd)        (test*)GetWindowWord((hwnd), 0)
#define test_SetPtr(hwnd, ptest)  (test*)SetWindowWord((hwnd), 0, (WORD)
(ptest))
#endif

// Default message handler

#define test_DefProc DefWindowProc

// Message handler functions, declared with a test* as their first argument,
// rather than an HWND.  Other than that, their signature is identical to
// that shown in WINDOWSX.H.
//
BOOL test_OnCreate(test* ptest, CREATESTRUCT FAR* lpcs);
void test_OnPaint(test* ptest);
```

```
//
// Code to register the test window class:
//
BOOL test_Init(HINSTANCE hinst)
{
    WNDCLASS cls;

    cls.hCursor         = ...;
    cls.hIcon           = ...;
    cls.lpszMenuName    = ...;
    cls.hInstance       = hinst;
    cls.lpszClassName   = "test";
    cls.hbrBackground   = ...;
    cls.lpfnWndProc     = test_WndProc;
    cls.style           = CS_DBLCLKS;
    cls.cbWndExtra      = sizeof(test*);  // room for instance data ptr
    cls.cbClsExtra      = 0;

    return RegisterClass(&cls);
}

// The window proc for class "test".  This demonstrates how instance data is
// attached to a window and passed to the message handler functions.
//
LRESULT CALLBACK test_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    test* ptest = test_GetPtr(hwnd);

    if (ptest == NULL)
    {
    // If we're creating the window, then try to allocate it.
    //
     if (msg == WM_NCCREATE)
     {
           // Create the instance data structure, set up the hwnd
backpointer
           // field, and associate it with the window.
           //
           ptest = (test*)LocalAlloc(LMEM_FIXED | LMEM_ZEROINIT,
sizeof(test));

           // If an error occured, return 0L to fail the CreateWindow call.
           // This will cause CreateWindow() to return NULL.
           //
           if (ptest == NULL)
           return 0L;
```

```
            ptest->hwnd = hwnd;
            test_SetPtr(hwnd, ptest);

            // NOTE: the rest of the test structure should be initialized
            // inside of Template_OnCreate() (or Template_OnNCCreate()).
Further
            // creation data may be accessed through the CREATESTRUCT FAR*
parameter.
            //
     }
     else
     {
            // It turns out WM_NCCREATE is NOT necessarily the first message
            // received by a top-level window (WM_GETMINMAXINFO is).
            // Pass messages that precede WM_NCCREATE on through to
            // test_DefProc
            //
            return test_DefProc(hwnd, msg, wParam, lParam);
     }
    }

    if (msg == WM_NCDESTROY)
    {
     LocalFree((HLOCAL)ptest);

            ptest = NULL;
     test_SetPtr(hwnd, NULL);

    }

    switch (msg)
    {
    HANDLE_MSG(ptest, WM_CREATE, test_OnCreate);
    HANDLE_MSG(ptest, WM_PAINT, test_OnPaint);
    ...

    default:
     return test_DefProc(hwnd, msg, wParam, lParam);
    }
}
```

Using Control Message Functions

The control message API functions perform a role that is the converse of message crackers: instead of handling messages sent to your window, they send messages to other windows (controls).

Each of the control message functions packs parameters into the appropriate *wParam*/*lParam* format and then calls **SendMessage**. These functions offer the same portability advantages that message crackers do; they free you from having to know how the current operating system packs *wParam* and *lParam*.

The function calls also improve readability of code and support better type checking. When used with the STRICT enhancements, the control message functions help prevent incorrect passing of message parameters.

To see how the control message functions work, first look at the following source code, which makes two calls to **SendMessage** to print all the lines in an edit control:

```
void PrintLines(HWND hwndEdit, WHND hwndDisplay)
{
    int line;
    int lineLast = (int)SendMessage(hwndEdit, EM_GETLINECOUNT, 0, 0L);

    for (line = 0; line < lineLast; line++)
    {
        int cch;
        char ach[80];

        *((LPINT)ach) = sizeof(ach);
        cch = (int)SendMessage(hwndEdit, EM_GETLINE,
                line, (LONG)(LPSTR)ach);

        PrintInWindow(ach, hwndDisplay);
    }
}
```

The source code below uses two control message functions, **Edit_GetLineCount** and **Edit_GetLine**, to perform the same task. This version of the code is shorter, easier to read, doesn't generate compiler warnings, and doesn't have any non-portable casts:

```
void PrintLines(HWND hwndEdit, WHND hwndDisplay)
{
    int line;
    int lineLast = Edit_GetLineCount(hwndEdit);

    for (line = 0; line < lineLast; line++)
    {
        int cch;
        char ach[80];

        cch = Edit_GetLine(hwndEdit, line, ach, sizeof(ach));

        PrintInWindow(ach, hwndDisplay);
    }
}
```

The control message API functions are listed in the table below. For more information, refer to the macro definitions in WINDOWSX.H and the documentation for the corresponding window message.

Control Message API Functions

Control Group	Functions
Static Text Controls:	**Static_Enable**(*hwnd*, *fEnable*)
	Static_GetIcon(*hwnd*, *hIcon*)
	Static_GetText(*hwnd*, *lpch*, *cchMax*)
	Static_GetTextLength(*hwnd*)
	Static_SetIcon(*hwnd*, *hIcon*)
	Static_SetText(*hwnd*, *lpsz*)
Button Controls:	**Button_Enable**(*hwnd*, *fEnable*)
	Button_GetCheck(*hwnd*)
	Button_GetState(*hwnd*)
	Button_GetText(*hwnd*, *lpch*, *cchMax*)
	Button_GetTextLength(*hwnd*)
	Button_SetCheck(*hwnd*, *check*)
	Button_SetState(*hwnd*, *state*)
	Button_SetStyle(*hwnd*, *style*, *fRedraw*)
	Button_SetText(*hwnd*, *lpsz*)
Edit Controls:	**Edit_CanUndo**(*hwnd*)
	Edit_EmptyUndoBuffer(*hwnd*)
	Edit_Enable(*hwnd*, *fEnable*)
	Edit_FmtLines(*hwnd*, *fAddEOL*)
	Edit_GetFirstVisible(*hwnd*)
	Edit_GetHandle(*hwnd*)
	Edit_GetLine(*hwnd*, *line*, *lpch*, *cchMax*)
	Edit_GetLineCount(*hwnd*)
	Edit_GetModify(*hwnd*)
	Edit_GetRect(*hwnd*, *lprc*)
	Edit_GetSel(*hwnd*)
	Edit_GetText(*hwnd*, *lpch*, *cchMax*)
	Edit_GetTextLength(*hwnd*)
	Edit_LimitText(*hwnd*, *cchMax*)
	Edit_LineFromChar(*hwnd*, *ich*)
	Edit_LineIndex(*hwnd*, *line*)
	Edit_LineLength(*hwnd*, *line*)
	Edit_ReplaceSel(*hwnd*, *lpszReplace*)
	Edit_Scroll(*hwnd*, *dv*, *dh*)
	Edit_SetHandle(*hwnd*, *h*)
	Edit_SetModify(*hwnd*, *fModified*)
	Edit_SetPasswordChar(*hwnd*, *ch*)
	Edit_SetRect(*hwnd*, *lprc*)
	Edit_SetRectNoPaint(*hwnd*, *lprc*)

Edit_SetSel(*hwnd*, *ichStart*, *ichEnd*)
Edit_SetTabStops(*hwnd*, *cTabs*, *lpTabs*)
Edit_SetText(*hwnd*, *lpsz*)
Edit_SetWordBreak(*hwnd*, *lpfnWordBreak*)
Edit_Undo(*hwnd*)

Scroll Bar Controls:
ScrollBar_Enable(*hwnd*, *flags*)
ScrollBar_GetPos(*hwnd*)
ScrollBar_GetRange(*hwnd*, *lpposMin*, *lpposMax*)
ScrollBar_SetPos(*hwnd*, *pos*, *fRedraw*)
ScrollBar_SetRange(*hwnd*, *posMin*, *posMax*, *fRedraw*)
ScrollBar_Show(*hwnd*, *fShow*)

List Box Controls:
ListBox_AddFile(*hwnd*, *lpszFilename*)
ListBox_AddItemData(*hwnd*, *data*)
ListBox_AddString(*hwnd*, *lpsz*)
ListBox_DeleteString(*hwnd*, *index*)
ListBox_Dir(*hwnd*, *attrs*, *lpszFileSpec*)
ListBox_Enable(*hwnd*, *fEnable*)
ListBox_FindItemData(*hwnd*, *indexStart*, *data*)
ListBox_FindString(*hwnd*, *indexStart*, *lpszFind*)
ListBox_GetAnchorIndex(*hwnd*)
ListBox_GetCaretIndex(*hwnd*)
ListBox_GetCount(*hwnd*)
ListBox_GetCurSel(*hwnd*)
ListBox_GetHorizontalExtent(*hwnd*)
ListBox_GetItemData(*hwnd*, *index*)
ListBox_GetItemHeight(*hwnd*, *index*) (1)
ListBox_GetItemRect(*hwnd*, *index*, *lprc*)
ListBox_GetSel(*hwnd*, *index*)
ListBox_GetSelCount(*hwnd*)
ListBox_GetSelItems(*hwnd*, *cItems*, *lpIndices*)
ListBox_GetText(*hwnd*, *index*, *lpszBuffer*)
ListBox_GetTextLen(*hwnd*, *index*)
ListBox_GetTopIndex(*hwnd*)
ListBox_InsertItemData(*hwnd*, *lpsz*, *index*)
ListBox_InsertString(*hwnd*, *lpsz*, *index*)
ListBox_ResetContent(*hwnd*)
ListBox_SelectItemData(*hwnd*, *indexStart*, *data*)
ListBox_SelectString(*hwnd*, *indexStart*, *lpszFind*)
ListBox_SelItemRange(*hwnd*, *fSelect*, *first*, *last*)
ListBox_SetAnchorIndex(*hwnd*, *index*)
ListBox_SetCaretIndex(*hwnd*, *index*)

ListBox_SetColumnWidth(*hwnd*, *cxColumn*)
ListBox_SetCurSel(*hwnd*, *index*)
ListBox_SetHorizontalExtent(*hwnd*, *cxExtent*)
ListBox_SetItemData(*hwnd*, *index*, *data*)
ListBox_SetItemHeight(*hwnd*, *index*, *cy*) (1)
ListBox_SetSel(*hwnd*, *fSelect*, *index*)
ListBox_SetTabStops(*hwnd*, *cTabs*, *lpTabs*)
ListBox_SetTopIndex(*hwnd*, *indexTop*)

Combo Box Controls:

ComboBox_AddItemData(*hwnd*, *data*)

ComboBox_AddString(*hwnd*, *lpsz*)
ComboBox_DeleteString(*hwnd*, *index*)
ComboBox_Dir(*hwnd*, *attrs*, *lpszFileSpec*)
ComboBox_Enable(*hwnd*, *fEnable*)
ComboBox_FindItemData(*hwnd*, *indexStart*, *data*)
ComboBox_FindString(*hwnd*, *indexStart*, *lpszFind*)
ComboBox_GetCount(*hwnd*)
ComboBox_GetCurSel(*hwnd*)
ComboBox_GetDroppedControlRect(*hwnd*, *lprc*) (1)
ComboBox_GetDroppedState(*hwnd*) (1)
ComboBox_GetEditSel(*hwnd*)
ComboBox_GetExtendedUI(*hwnd*) (1)
ComboBox_GetItemData(*hwnd*, *index*)
ComboBox_GetItemHeight(*hwnd*)
ComboBox_GetLBText(*hwnd*, *index*, *lpszBuffer*)
ComboBox_GetLBTextLen(*hwnd*, *index*)
ComboBox_GetText(*hwnd*, *lpch*, *cchMax*)
ComboBox_GetTextLength(*hwnd*)
ComboBox_InsertItemData(*hwnd*, *index*, *data*)
ComboBox_InsertString(*hwnd*, *index*, *lpsz*)
ComboBox_LimitText(*hwnd*, *cchLimit*)
ComboBox_ResetContent(*hwnd*)
ComboBox_SelectItemData(*hwnd*, *indexStart*, *data*)
ComboBox_SelectString(*hwnd*, *indexStart*, *lpszSelect*)
ComboBox_SetCurSel(*hwnd*, *index*)
ComboBox_SetEditSel(*hwnd*, *ichStart*, *ichEnd*)
ComboBox_SetExtendedUI(*hwnd*, *flags*) (1)
ComboBox_SetItemData(*hwnd*, *index*, *data*)
ComboBox_SetItemHeight(*hwnd*, *cyItem*) (1)
ComboBox_SetText(*hwnd*, *lpsz*)

ComboBox_ShowDropdown(*hwnd*, *fShow*)

1 Supported only for Win32, not for Windows 3.*x*. These functions are not available if you define the symbol WINVER as equal to 0x0300, on the command line or with a **#define** statement.

Writing Portable C Programs

Because C compilers exist on a variety of computers, some C applications developed for one computer system can be ported to other systems. However, some aspects of language behavior depend on how a particular C compiler is implemented and how a specific computer operates. Therefore, when designing a program to be ported to another system, it is important that you examine programming assumptions. This guide describes those assumptions that can affect writing portable programs.

The American National Standards Institute Standard for the C Language (the ANSI Standard) details every instance where language behavior is defined by the implementation.

Assumptions About Hardware

To make C programs portable, you must examine two aspects of your code: hardware assumptions and compiler dependency. This guide deals with hardware assumptions. Assumptions About the Compiler deals with compiler dependency.

Size of Basic Types

In C, the size of basic types (**char**, **signed int**, **unsigned int**, **float**, **double**, and **long double**) is implementation-defined, so relying on a particular data type to be a given size reduces the portability of a program.

Because the size of basic types is left to the implementation, do not make assumptions about the size or alignment of data types within aggregate types. You should always use the **sizeof** operator to determine the size or amount of storage required for a variable or a type.

The following topics discuss rules governing the size of specific data types.

- Type char
- Type int and Type short int
- Type float, Type double, and Type long double
- Microsoft C Type Sizes

Type Char

Type **char** is the smallest of the basic types, but it must be large enough to hold any of the characters in the implementation's basic character set. Normally, variables of type **char** are 1 byte.

Type Int and Type Short Int

Type **int** often corresponds to the register size of the target computer. Type **short int** may be less than or equal to the size of type **int**. Both **int** and **short** are greater than or equal to the size of type **char** but less than or equal to the size of type **long**.

If you assume that type **int** is a certain size, your code may not be portable because:

- An **int** can be defined as a 16-bit (2-byte) or a 32-bit quantity.
- An **int** is not always large enough to hold array indexes. For large arrays, you must use **unsigned int**; for extremely large arrays, use **long** or **unsigned long**. To be certain your code is portable, define your array indexes as type **size_t**. You may not know, before porting your code, the maximum value to expect an array index of type **int** to hold. The file LIMITS.H contains manifest constants, listed below, for the maximum and minimum values of each basic integral type.

Constant	Value
CHAR_BIT	Number of bits in a variable of type **char**
CHAR_MIN	Minimum value a variable of type **char** can hold
CHAR_MAX	Maximum value a variable of type **char** can hold
SCHAR_MIN	Minimum value a variable of type **signed char** can hold
SCHAR_MAX	Maximum value a variable of type **signed char** can hold
UCHAR_MAX	Maximum value a variable of type **unsigned char** can hold
SHRT_MIN	Minimum value a variable of type **short** can hold
SHRT_MAX	Maximum value a variable of type **short** can hold
USHRT_MAX	Maximum value a variable of type **unsigned short** can hold
INT_MIN	Minimum value a variable of type **int** can hold
INT_MAX	Maximum value a variable of type **int** can hold
UINT_MAX	Maximum value a variable of type **unsigned int** can hold
LONG_MIN	Minimum value a variable of type **long** can hold
LONG_MAX	Maximum value a variable of type **long** can hold
ULONG_MAX	Maximum value a variable of type **unsigned long** can hold

Type Float, Type Double, and Type Long Double

Type **float** is the smallest of the basic floating-point types. Type **double** is usually larger than type **float**, and type **long double** is usually the largest of the floating-point types. You can make the following portability assumptions about floating-point types:

- Any value that can be represented as type **float** can be represented as type **double** (type **float** is a subset of type **double**).
- Any value that can be represented as type **double** can be represented as type **long double** (type **double** is a subset of type **long double**).

The file FLOAT.H contains manifest constants, listed below, for the maximum and minimum values of each basic floating-point type.

Constant	Value
DBL_DIG	Number of decimal digits of precision a variable of type **double** can hold
DBL_MAX	Maximum value a variable of type **double** can hold
DBL_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type **double** can hold
DBL_MAX_EXP	Maximum value (base 2) the exponent of a variable of type **double** can hold
DBL_MIN	Minimum positive value a variable of type **double** can hold
DBL_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type **double** can hold
DBL_MIN_EXP	Minimum value (base 2) the exponent of a variable of type **double** can hold
FLT_DIG	Number of decimal digits of precision a variable of type **float** can hold
FLT_MAX	Maximum value a variable of type **float** can hold
FLT_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type **float** can hold
FLT_MAX_EXP	Maximum value (base 2) the exponent of a variable of type **float** can hold
FLT_MIN	Minimum positive value a variable of type **float** can hold
FLT_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type **float** can hold
FLT_MIN_EXP	Minimum value (base 2) the exponent of a variable of type **float** can hold
LDBL_DIG	Number of decimal digits of precision a variable of type **long double** can hold
LDBL_MAX	Maximum value a variable of type **long double** can hold
LDBL_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type **long double** can hold
LDBL_MAX_EXP	Maximum value (base 2) the exponent of a variable of type **long double** can hold
LDBL_MIN	Minimum positive value a variable of type **long double** can hold

LDBL_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type **long double** can hold
LDBL_MIN_EXP	Minimum value (base 2) the exponent of a variable of type **long double** can hold

Microsoft C Type Sizes

The following table summarizes the size of the basic types in Microsoft C:

Size of Basic Types in Microsoft C

Type	Number of bytes
char, **unsigned char**	1
short, **unsigned short**	2
int, **unsigned int**	2 or 4 (*)
near pointer	2 or 4 (*)
long, **unsigned long**	4
far pointer	4
float	4
double	8
long double	10

* These data types have different sizes in 16- and 32-bit environments.

Storage Order and Alignment

The C language does not define any specific layout for the storage of data items relative to one another. The layout for storage of structure elements, or unions within a structure or union, is defined by the implementation.

Some processors require that data longer than 1 byte be aligned to 2-byte or 4-byte boundaries. Other processors, such as the 80*x*86 family, do not have such a restriction. However, the 80*x*86 processors work more efficiently with aligned data.

Most RISC processors expect each piece of data in memory to be aligned on a boundary appropriate to its size. For example, an *n*-byte integer can be aligned on a boundary whose address is a multiple of *n*-bytes, up to a maximum of 8 bytes. This restriction permits the memory system to run much faster.

Ordinarily, alignment has no effect on correctly written programs because the compiler inserts unused space ("padding") between variables wherever necessary to conform to the rules.

Structure Order and Alignment

The following example illustrates how alignment can affect your program. In the example, a structure is cast to type **long** because the programmer knew the order in which a particular implementation stored data.

```
/* Nonportable code */
struct time
{
    char hour;      /* 0 < hour < 24   — fits in a char */
    char minute;    /* 0 < minute < 60 — fits in a char */
    char second;    /* 0 < second < 60 — fits in a char */
};

    .
    .
    .
    struct time now, alarm_time;
    .
    .
    .
    if ( *(long *)&now >= *(long *)&alarm_time )
    {
        /* sound an alarm */
    }
```

The preceding code makes these nonportable assumptions:

- The data for *hour* will be stored in a higher order position than *minute* or *second*. Because C does not guarantee storage order or alignment of structures or unions, the code may not be portable to other computers.
- Three variables of type **char** will be shorter than or the same length as a variable of type **long**. Thus, the code is not portable according to the rules governing the size of basic types, as described in Size of Basic Types.

If either of these assumptions proves false, the comparison (**if** statement) is invalid.

To make the program in the preceding example portable, you can break the comparison between the two long integers into a component-by-component comparison. This technique is illustrated in the following example:

```
/* Portable code */
struct time
{
    char hour;      /* 0 < hour < 24   — fits in a char */
    char minute;    /* 0 < minute < 60 — fits in a char */
```

```
    char second;   /* 0 < second < 60 — fits in a char */
};

    .
    .
    .
    struct time now, alarm_time;
    .
    .
    .
    if ( time_cmp( now, alarm_time ) >= 0 )
    {
        /* sound an alarm */
    }
    .
    .
    .

int time_cmp( struct time t1, struct time t2 )
{
    if( t1.hour != t2.hour )
        return( t2.hour - t1.hour );
    if( t1.minute != t2.minute )
        return( t2.minute - t1.minute );
    return( t2.second - t1.second );
}
```

Even a program that follows the rules given previously may have trouble when writing data on one computer and reading it on another. In addition to padding, computers differ with regard to endianness, floating-point formats, and size of data types. If I/O speed is not important, a possible solution is to use ASCII files rather than binary ones.

Union Order and Alignment

Programmers use unions most often for two purposes: to store data whose exact type is not known until run time or to access the same data in different ways.

Unions falling into the second category are usually not portable. For example, the following union is not portable:

```
union tag_u
{
     char bytes_in_long[4];
     long a_long;
};
```

The intent of the preceding union is to access the individual bytes of a variable of type **long**. However, the union may not work as intended when ported to other computers because:

- It relies on a constant size for type **long**.
- It may assume byte ordering within a variable of type **long**. (Byte ordering is described in detail in Byte Order in a Word.)

Byte Order in a Word

The order of bytes within an integral type longer than a byte (**short**, **int**, or **long**) can vary among computers. Computers that number the bytes from left to right with the least-significant byte within a word being byte zero are called "little-endian." Computers that number the bytes from left to right with the least-significant byte being byte 3 are called "big-endian." Code that assumes an internal order is not portable, as shown by this example:

```
/*
 * Nonportable structure to access an int in bytes.
 */
struct tag_int_bytes
{
    char lobyte;
    char hibyte;
};
```

A more portable way to access the individual bytes in a word is to define two macros that rely on the constant **CHAR_BIT**, defined in LIMITS.H:

```
#define LOBYTE(a) (char)((a) & 0xff)
#define HIBYTE(a) (char)((unsigned)(a) >> CHAR_BIT)
```

The **LOBYTE** macro is still not completely portable. It assumes that a **char** is 8 bits long, and it uses the constant **0xff** to mask the high-order 8 bits. Because portable programs cannot rely on a given number of bits in a byte, consider the revision below:

```
#define LOBYTE(a) (char)((a) & ((unsigned)~0>>CHAR_BIT))
#define HIBYTE(a) (char)((unsigned)(a) >> CHAR_BIT)
```

This revised **LOBYTE** macro performs a bitwise complement on 0; that is, all zero bits are turned into ones. The macro then takes that unsigned quantity and shifts it right far enough to create a mask of the correct length for the implementation.

The following code assumes that the order of bytes in a word will be least-significant first:

```
int c;
.
.
.
fread( &c, sizeof( char ), 1, fp );
```

The code attempts to read one byte as an **int**, without converting it from a **char**. However, the code will fail in any implementation where the low-order byte is not the first byte of an **int**. The following solution is more portable. In this example, the data is read into an intermediate variable of type **char** before being assigned to the integer variable:

```
int c;
char ch;
.
.
.
fread( &ch, sizeof( char ), 1, fp );
c = ch;
```

The following example shows how to use the C run-time function **fgetc** to return the value. The **fgetc** function returns type **char**, but the value is promoted to type **int** when it is assigned to a variable of

type **int**:

```
int c;
.
.
.
c = fgetc( fp );
```

You might create porting problems by placing small objects side by side to make a bigger object, or splitting a big object into several small objects. For example, the following code that reads and compares a pair of short integers is computer-dependent because on some computers the 0th element of the array represents the high-order half of the word rather than the low-order half:

```
char carr[BUFSIZ];
err = read( 0, carr, 4 );
if ( ( carr[0] | ( carr[1] << 8 ) ) > ( carr[2] | ( carr[3] << 8 ) )
...
```

There is never a problem if you use the correct data type and let the compiler deal with the order of the bytes:

```
short sarr[BUFSIZ];
err = read( 0, (char *) sarr, 2 * sizeof( short ) );
if ( sarr[0] > sarr[i] )
...
```

Microsoft C Specific

Microsoft C normally aligns data types longer than one byte to an even-byte address for improved performance. See the **/Zp** compiler option.

Floating-Point Accuracy

Some compilers use extended precision even when your program does not specify it because it is more natural for the design of the processor to use the extended precision. When an expression is evaluated using extended precision, you may get a slightly different answer than if it were evaluated in double precision. Intel 80*x*87 math coprocessors use 80-bit precision for calculations, whereas RISC processors often use 64-bit precision.

In addition to differences resulting from hardware, each implementation typically differs in the algorithms and characteristics of its math library. Even IEEE computers that are otherwise identical may produce different results due to differences in math libraries.

Reading and Writing Structures

Many C programs read data from disk into structures and write data to disk from structures. The functions that perform disk I/O in C require you to specify the number of bytes to be transferred. You should always use the **sizeof** operator to obtain the size of the data to be read or written, because differing data type sizes or alignment schemes may alter the size of a given structure. The following code shows one such use of the **sizeof** operator:

```
fread( &my_struct, sizeof(my_struct), 1, fp );
```

Microsoft C Specific

When performing disk input and output in Microsoft C, structures may be different sizes depending on the structure-packing option you have selected. See the **/Zp** compiler option.

Bit Fields in Structures

The Microsoft C compiler implements bit fields. However, many C compilers do not.

Bit fields allow you to access the individual bits within a data item. While the practice of accessing the bits in a data item is inherently nonportable, you can improve your chances of porting a program that uses bit fields if you make no assumptions about order of assignment, or size and alignment of bit fields.

Order of Assignment

Because the order of assignment of bit fields in memory is left to the implementation, you cannot rely on a particular entry in a bit field structure to be in a higher order position than another. (This problem is similar to the portability constraint imposed by alignment of basic data types in structures. The C language does not define any specific layout for the storage of data items relative to one another.) See Storage Order and Alignment for more information.

Size and Alignment of Bit Fields

The Microsoft C compiler supports bit fields up to the size of the type **long**. Each individual member of the bit field structure can be up to the size of the declared type. Some compilers do not support bit field structure elements that are longer than type **int**.

The following example defines a bit field, **short_bitfield**, that is shorter than type **int**:

```
struct short_bitfield
{
    unsigned usr_bkup : 1; /* 0  <= usr_bkup <  1 */
    unsigned usr_sec  : 4; /* 9 <= usr_sec < 16 */
};
```

The following example defines a bit field, **long_bitfield**, that has an element longer than type **int** in a 16-bit environment:

```
struct long_bitfield
{
    unsigned long disk_pos : 22;  /* 0 <= disk_pos   < 4,194,304 */
    unsigned long rec_no   : 10;  /* 0 <= rec_no < 1,024 */
};
```

The bit field **short_bitfield** is likely to be supported by more implementations than **long_bitfield**.

Microsoft C Specific

The following example introduces another portability issue: alignment of data defined in bit fields.

```
struct long_bitfield
{
    unsigned int day    : 5;  /* 0 <= day   < 32 */
    unsigned int month  : 4;  /* 0 <= month < 16 */
    unsigned int year   : 11; /* 0 <= year  < 2048 */
};
```

In the 32-bit environment, all three elements can fit within a single 32-bit **int**, so there is no gap between any of the elements in Microsoft C's representation of the structure. Microsoft C for the 16-bit environment, by contrast, uses a word size of 16 bits. Because the compiler does not allow structure elements to cross a word boundary, there is a 7-bit gap between the second and third elements in the 16-bit environment.

Processor Arithmetic Mode

Two types of arithmetic are common on digital computers: one's-complement arithmetic and two's-complement arithmetic. Some programs assume that all target computers perform two's-complement arithmetic. If you take advantage of the fact that a given operation causes a particular bit pattern to be set on either a one's-complement or two's-complement computer, your program will not be portable. For example, two's-complement computers represent the 8-bit integer value -1 as a binary 11111111. A one's-complement computer represents the same decimal value (-1) as 11111110. Some programmers assume that -1 will fill a byte or a word with ones, and they use it to construct a mask template that they later shift. This will not work correctly on one's-complement computers, but the error will not surface until the least-significant bit is used.

In two's-complement arithmetic, there is only one value that represents zero. In one's-complement arithmetic, there is a value for zero and a value for negative zero. Use the C relational operators to handle this anomaly correctly; if you write code that deliberately circumvents the C relational operators, tests for zero or NULL may not operate correctly.

Microsoft C Specific

Microsoft C uses two's-complement arithmetic exclusively.

Pointers

One of the most powerful but potentially dangerous features of the C language is its use of indirect addressing through pointers. Bugs introduced by misusing pointers can be difficult to detect and isolate because the error often corrupts memory unpredictably.

Casting Pointers

Be sure you do not make nonportable assumptions when casting pointers to different types. The following code is nonportable because using a cast to change an array of **char** to a pointer of type **long** assumes a particular byte-ordering scheme. This is discussed in greater detail in Byte Order in a Word.

```
/* Nonportable coercion */
char c[4];
long *lp;

lp = (long *)c;
*lp = 0x12345678L;
```

Pointer Size

A pointer can be assigned (or cast) to any integer type large enough to hold it, but the size of the integer type depends on the computer and the implementation. Therefore, you cannot assume that a pointer is the same size as an integer; that is:

```
sizeof( char * ) != sizeof( int )
```

To determine the size of any unmodified data pointer, use:

```
sizeof( void * )
```

This expression returns the size of a generic data pointer.

Pointer Subtraction

Code that assumes that pointer subtraction yields an **int** value is nonportable. Pointer subtraction yields a result of type **ptrdiff_t** (defined in STDDEF.H). Portable code must always use variables of type **ptrdiff_t** for storing the result of pointer subtraction.

The Null Pointer

In most implementations, NULL is defined as 0. In Microsoft C, it is defined as **((void *)0)**. Because code pointers and data pointers are often different sizes, using 0 for the null pointer for both can lead to nonportability. The difference in size between code pointers and data pointers causes problems for functions that expect pointer arguments longer than an **int**. To avoid these problems, use the null pointer, as defined in the include file STDDEF.H; use prototypes; or explicitly cast NULL to the correct data type. Here is a portable way to use the null pointer:

```
/* Portable use of the null pointer */
main()
{
    func1( (char *)NULL );
    func2( (void *(*)())NULL );
}

void func1( char * c )
{
}

void func2( void *(* func)() )
{
}
```

The invocations of **func1** and **func2** explicitly cast NULL to the correct size. In the case of **func1**, NULL is cast to type **char** *; in the case of **func2**, it is cast to a pointer to a function that returns type **void**.

Microsoft C Specific

In the 32-bit environment, support for memory models and pointers of different sizes (**__near**, **__far**, and **__huge**) is removed. All pointers are 32 bits in size, and an attempt to use near pointers causes an error. You can, however, use the **/Zf** option to cause the compiler to simply ignore **__far** pointers.

Address Space

The amount of available memory and the address space on systems varies, depending on many factors outside your control. A program designed with portability in mind should handle insufficient-memory situations. To ensure that your program handles these situations, you should always check the error return from any of the dynamic-memory-allocation routines, such as **malloc**, **calloc**, **strdup**, and **realloc**.

Character Set

The C language does not define the character set used in an implementation. This means that any programs that assume the character set to be ASCII are nonportable.

Character Classification

The standard C run-time support contains a complete set of character classification macros and functions. These functions are defined in the CTYPE.H file and are guaranteed to be portable:

isalnum

isalpha

iscntrl

isdigit

isgraph

islower

isprint

ispunct

isspace

isupper

isxdigit

The following code fragment is not portable to implementations that do not use the ASCII character set:

```
/* Nonportable */
if( c >= 'A' && c <= 'Z' )
    /* uppercase alphabetic */
```

Instead, consider using this:

```
/* Portable */
if( isalpha(c) && isupper(c) )
    /* uppercase alphabetic */
```

The first example is nonportable because it assumes that uppercase A is represented by a smaller value than uppercase Z and that no lowercase characters fall between the values of A and Z. The second example is portable because it uses the character classification functions to perform the tests.

In a portable program, you should not perform any comparison on variables of type **char** except strict equality (**==**). You cannot assume the character set follows an increasing sequence – that may not be true on a different computer.

Case Translation

Translation of characters from uppercase to lowercase or from lowercase to uppercase is called *case translation.* The following example shows a coding technique for case translation not portable to implementations using a non-ASCII character set:

```
#define make_upper(c) ((c)&0xcf)
#define make_lower(c) ((c)|0x20)
```

This code takes advantage of the fact that you can map uppercase to lowercase simply by changing the state of bit 6. It is extremely efficient but nonportable. To write portable code, use the case-translation macros **toupper** and **tolower** (defined in CTYPE.H).

Assumptions About the Compiler

Different compilers translate C source code into object code in different ways. The ANSI draft standard for the C programming language defines how many of these translations must be done; others are implementation-defined.

This guide describes assumptions about how the compiler translates your C code, which can make your programs nonportable.

Sign Extension

"Sign extension" is the propagation of the sign bit to fill unoccupied space when promoting to a more significant type or when performing bitwise right-shift operations.

Promotion from Shorter Types

Integral promotions from shorter types occur when you make an assignment, perform arithmetic, perform a comparison, or perform an explicit cast.

The behavior of integral promotion is well defined, except for type **char**. The implementation defines whether type **char** is treated as signed or unsigned. The following code fragment is an example of promotion as a result of assignment:

```
char c1 = -3;
int  i1;

i1 = c1;
```

In this example, the expected result of the assignment statement is that **i1** will be set to -3. If the implementation defines type **char** as unsigned, however, sign extension will not occur, and **i1** will be 253 (on a two's-complement computer).

Promotion can also occur as a result of a comparison of different types:

```
char c;

if( c == 0x80 )
    .
    .
    .
```

This comparison never evaluates as true on an implementation that sign-extends **char** types but treats hexadecimal constants as unsigned. Use a character constant of the form '\x80', or explicitly cast the constant to type **char** to perform the comparison correctly.

The following comparison, which is an example of promotion as a result of a cast, is also nonportable:

```
char c;
unsigned int u;

if( u == (unsigned)c )
```

There are two problems with this code:

- The **char** type may be treated as signed or unsigned, depending on the implementation.
- If the **char** type is treated as signed, it can be converted to **unsigned** in two ways: the **char** value may first be sign-extended to **int**, then converted to **unsigned**; or the **char** may be converted to **unsigned char**, then sign-extended to **int** length.

It is always safe to compare a **signed int** with a **char** constant because C requires all character constants to be positive.

Variables of type **char** are promoted to type **int** when passed as arguments to a function. This causes sign extension on some computers. Consider the following code:

```
char c = 128;

printf( "%d\n", c );
```

Microsoft C Specific

Microsoft C allows you to treat type **char** as signed or unsigned. By default, a **char** is considered signed, but if you change the default **char** type using the **/J** compiler option, you can treat it as

unsigned.

Bitwise Right-Shift Operations

Positive or unsigned integral types (**char**, **short**, **int**, and **long**) yield positive or zero values after a bitwise right-shift (**>>**) operation. For example,

```
(char)120 >> 4
```

yields 7,

```
(unsigned char)240 >> 8
```

yields 0,

```
(int)500 >> 8
```

yields 1, and

```
(unsigned int)65535 >> 4
```

yields 4,095.

Negative-signed integral types yield implementation-defined values after a bitwise right-shift operation. This means that you must know whether you want to do a signed or unsigned shift, then code accordingly.

If you don't know how the implementation performs, you might get unexpected results. For example, **(signed char)0x80 >> 3** yields 0xf0 if the implementation performs sign extension on right bitwise shifts. If the implementation does not perform the sign extension, the result is 0x10.

You can use right shifts to speed up division when the divisor can be represented by powers of 2 and the dividend is positive. To maintain portability, you should use the division operator.

To perform an unsigned shift, explicitly cast the data to an unsigned type. To perform a shift that extends the sign bit, use the division operator as follows: divide by 2^ *(n)* , where *n* is the number of bits you want to shift.

Length and Case of Identifiers

Some implementations do not support long identifiers. Some allow only 6 characters, while others allow as many as 32. They may report each identifier that exceeds the maximum length or truncate identifiers to a given length. Truncation causes serious problems, especially if you have a number of similarly named variables within the scope of a block of code, such as the following:

```
double acct_receivable_30_days;
double acct_receivable_60_days;
double acct_receivable_90_days;
double current_interest_rate;

acct_receivable_30_days *= current_interest_rate;
```

If your target system retains only six significant characters, you need to rename all your *acct_receivable* variables.

Case sensitivity also affects portability. C is usually a case-sensitive language. That is, **CalculateInterest** is not considered the same identifier as **calculateinterest**. Some systems are not case sensitive, however, so to write portable code, differentiate your identifiers by something other than case.

These problems with identifiers can occur in two locations: the compiler and the linker or loader. Even if the compiler can handle long and case-differentiated identifiers, if the linker or loader cannot, you can get duplicate definitions or other unexpected errors.

Microsoft C Specific

The Microsoft C compiler issues the **/NOIGNORECASE** command to the Microsoft Linker (LINK), which directs it to consider case significant.

Register Variables

The number and type of register variables in a function depend on the implementation. You can declare more variables as **register** than the number of physical registers the implementation uses. In such a case, the compiler treats the excess register variables as **automatic**.

Because the types that qualify for **register** class differ among implementations, invalid **register** declarations are treated as **automatic**.

If you declare variables as **register** to optimize performance, declare them in decreasing order of importance to make sure the compiler allocates a register to the most important variables.

Microsoft C Specific

The compiler ignores **register** declarations if you select the global register allocation optimization. You can select global register allocation as follows:

Environment	Selection
CL command line	Specify either the **/Oe** or **/Ox** option.
pragma	Use the **optimize** pragma with the **e** parameter.

Evaluation Order

The C language does not guarantee the evaluation order of most expressions. Avoid writing constructs that depend on evaluation within an expression to proceed in a particular manner. For example,

```
i = 0;
func( i++, i++ );
.
.
.
func( int a, int b )
{
```

A compiler could evaluate this code fragment and pass 0 as a and 1 as b. It could also pass 1 as a and 0 as b and conform equally with the standards.

The C language does guarantee that an expression will be completely evaluated at any given "sequence point." A sequence point is a point in the syntax of the language at which all side effects of an expression or series of expressions have been completed.

These are the sequence points in the C language:

- The semicolon (**;**) statement separator
- The call to a function after the arguments have been evaluated
- The end of the first operand of one of the following:
  - Logical AND (**&&**)
  - Logical OR (**||**)
  - Conditional (**?**)
  - Comma separator (**,**) when used to separate statements or in expressions; the comma separator is not a sequence point when it is used between variables in declaration statements or between parameters in a function invocation
- The end of a full expression, such as:
  - An initializer
  - The expression in an expression statement (for example, any expression inside parentheses)
  - The controlling expression of a **while** or **do** statement
  - Any of the three expressions of a **for** statement
  - The expression in a **return** statement

Function and Macro Arguments with Side Effects

Run-time support functions can be implemented either as functions or as macros. Avoid including expressions with side effects inside function invocations unless you are sure the function will not be implemented as a macro. Here is an illustration of how an argument with side effects can cause problems:

```
#define limit_number(x) ((x > 1000) ? 1000 : (x))

a = limit_number( a++ );
```

If a is greater than 1000, it is incremented once. If a is less than or equal to 1000, it is incremented twice, which is probably not the intended behavior.

A macro can be used safely with an argument that has side effects if it evaluates its parameter only once. You can determine whether a macro is safe only by inspecting the code.

A common example of a run-time support function that is often implemented as a macro is **toupper**. You will find your program's behavior confusing if you use the following code:

```
char c;

c = toupper( getc() );
```

If **toupper** is implemented as a function, **getc** will be called only once, and its return value will be translated to uppercase. However, if **toupper** is implemented as a macro, **getc** will be called once or twice, depending on whether **c** is uppercase or lowercase. Consider the following macro example:

```
#define toupper(c) ( (islower(c)) ? _toupper(c) : (c) )
```

If you include the **toupper** macro in your code, the preprocessor expands it as follows:

```
/* What you wrote */
c = toupper( getc() );

/* Macro expansion */
ch = (islower( (getc()) ) ? _toupper( getc() ) : (getc()) );
```

The expansion of the macro shows that the argument to toupper will always be called twice: once to determine if the character is lowercase and the next time to perform case translation (if necessary). In the example, this double evaluation calls the **getc** function twice. Because **getc** is a function whose side effect is to read a character from the standard input device, the example requests two characters from standard input.

Environment Differences

Many programs perform some file I/O. When writing these programs for portability, consider the following:

- Do not hard code filenames or paths. Use constants you define either in a header file or at the beginning of the program.
- Do not assume the use of any particular file system.
- Do not assume a particular display size (number of rows and columns).
- Do not assume that display attributes exist. Some environments do not support such attributes as color, underlined text, blinking text, highlighted text, inverse text, protected text, or dim text.

Microsoft C Byte Ordering

For all Windows programming, the processor runs in "little endian" mode. If b0 represents a less significant byte than b1, the layout of a short will be b0 b1, and the layout of a long will be b0 b1 b2 b3. This is in contrast to "big endian" mode where a short would be b1 b0, and a long would be b3 b2 b1 b0.

WINDOWS.H and STRICT Type Checking

The WINDOWS.H file contains a number of type definitions, macros, and structures that aid in writing source code portable between versions of Microsoft Windows. Some of the WINDOWS.H features are enabled when the STRICT symbol is defined on the command line or makefile. This guide explains how these STRICT features affect the writing of correct code and what the advantages of using them are.

This guide discusses the following major topics:

New Types and Macros

Porting 16-bit Code to 32-bit Windows introduced some new standard types for Windows programming. Use of the old types, such as **FAR PASCAL** for declaring Windows procedures, may work in existing code but is not guaranteed to work in all future versions of Windows. Therefore, you should convert your code to use the new standards wherever appropriate.

General Data Types

The following table summarizes the new standard types defined in WINDOWS.H. These types are polymorphic (they can contain different kinds of data) and are useful generally throughout applications. There are also other new types, handles, and function pointers that are introduced in following sections.

Typedef	Description
WINAPI	Use in place of **FAR PASCAL** in function declarations. If you are writing a DLL with exported function entry points, you can use this for your own functions.
CALLBACK	Use in place of **FAR PASCAL** in application callback routines such as window procs and dialog procs.
LPCSTR	Same as **LPSTR**, except used for read-only string pointers. Defined as (**const char FAR***).
UINT	Portable unsigned integer type whose size is determined by host environment (32 bits for Windows NT). Synonym for **unsigned int**. Used in place of **WORD** except in the rare cases where a 16-bit unsigned quantity is desired even on 32-bit platforms.
LRESULT	Type used for declaration of Window procedure return value.
WPARAM	Type used for declaration of first general purpose Window procedure parameter.
LPARAM	Type used for declaration of second general purpose Window procedure parameter.
LPVOID	Generic pointer type, equivalent to **(void *)**. Should be used in preference to **LPSTR**.

Utility Macros

WINDOWS.H provides a series of utility macros that are useful for working with the types listed in the previous section. These macros, listed in the following table, help create and extract data from these types. The **FIELDOFFSET** macro is particularly useful when you need to give the numeric offset of a structure member as an argument.

Utility	Description
MAKELPARAM(*low*, *high*)	Combines two 16-bit quantities into an **LPARAM**.
MAKELRESULT(*low*, *high*)	Combines two 16-bit quantities into an **LRESULT**.
MAKELP(*sel*, *off*)	Combines a selector and an offset into a **FAR VOID*** pointer. Useful only for Windows 3.*x*.
SELECTOROF(*lp*)	Extracts the selector part of a far pointer. Returns a **UINT**. Useful only for Windows 3.*x*.
OFFSETOF(*lp*)	Extracts the offset part of a far pointer. Returns a **UINT**. Useful only for Windows 3.*x*.
FIELDOFFSET(*type*, *field*)	Calculates the offset of a member of a data structure. The *type* is the type of structure, and *field* is the name of the structure member or field.

New Handle Types

In addition to the existing Windows handle types such as **HWND**, **HDC**, **HBRUSH**, and so on, WINDOWS.H defines the following new handle types. They are particularly important if STRICT type checking is enabled, but you can use these even if you do not define STRICT.

Handle	Description
HINSTANCE	Instance handle type
HMODULE	Module handle type
HBITMAP	Bitmap handle type
HLOCAL	Local handle type
HGLOBAL	Global handle type
HTASK	Task handle type
HFILE	File handle type
HRSRC	Resource handle type
HGDIOBJ	Generic GDI object handle type (except HMETAFILE)
HMETAFILE	Metafile handle type
HDWP	**DeferWindowPos**() handle
HACCEL	Accelerator table handle
HDRVR	Driver handle (Windows NT only)

Using STRICT to Improve Type Checking

Defining the STRICT symbol enables features that require you to be more careful in declaring and using types. This may sound like a burden, but it is actually an aid to writing more portable code. This extra care will also reduce the time you spend debugging. Enabling STRICT redefines certain data types so that the compiler won't permit assignment from one type to another without an explicit cast. This is especially helpful with Windows code. Errors in passing data types are reported at compile time instead of causing fatal errors at run time.

When STRICT is defined, WINDOWS.H type definitions change as follows:

- Specific handle types are defined so as to be mutually exclusive; for example, you won't be able to pass an **HWND** where an **HDC** type argument is required. Without STRICT, all handles are defined as integers, so the compiler doesn't prevent you from using one type of handle where another type is expected.
- All callback function types (dialog procedures, window procedures, and hook procedures) are defined with full prototypes. This prevents you from declaring callback functions with incorrect parameter lists.
- Parameter and return value types that should use a generic pointer are declared correctly as **LPVOID** instead of as **LPSTR** or other pointer type.
- The **COMSTAT** structure is now declared according to the ANSI standard.

Enabling STRICT Type Checking

To enable STRICT type checking, just define the symbol name "STRICT". You can specify this definition on the command line or in a makefile by giving **/DSTRICT** as a compiler option.

To define STRICT on a file-by-file basis (supported by C but not C++ as explained in the note in this section), insert a **#define** statement before including WINDOWS.H in files where you want to enable STRICT:

```
#define STRICT
#include WINDOWS.H
```

For best results, you should also set the warning level for error messages to at least /W3. This is good policy with Windows applications in any case, because a coding practice that causes a warning (for example, passing the wrong number of parameters) usually causes a fatal error at run time if it is not corrected.

Note If you are writing a C++ application, you don't have the option of applying STRICT to only some of your source files. Because of the way C++ "type safe linking" works, mixing STRICT and non-STRICT source files in your application can cause linking errors.

Making Your Application STRICT Compliant

Some source code that in the past compiled successfully might produce error messages when you enable STRICT type checking. The following sections describe the minimal requirements you need to follow, where applicable, to make sure your code compiles when STRICT is enabled. There are other steps not strictly required but recommended, especially if you want to produce portable code. These are covered in Making Best Use of STRICT Type Checking.

General Requirements

The principal requirement is that you must declare correct handle types and function pointers instead of relying on more general types such as **unsigned int** and **FARPROC**. You cannot use one handle type where another is expected. This requirement also means that you may have to change function declarations and use more type casts.

For best results, the generic **HANDLE** type should not be used except where necessary. Consult New Types and Macros for a list of new specific handle types.

Using Function Pointers

Always declare function pointers with the proper function type (such as **DLGPROC** or **WNDPROC**) rather than **FARPROC**. You'll need to cast function pointers to and from the proper function type when using **MakeProcInstance**, **FreeProcInstance**, and other functions that take or return a **FARPROC**, as shown in the following code:

```
BOOL CALLBACK DlgProc(HWND hwnd, UINT msg, WPARAM wParam,
                                  LPARAM lParam);
DLGPROC lpfnDlg;

lpfnDlg = (DLGPROC)MakeProcInstance(DlgProc, hinst);
...
FreeProcInstance((FARPROC)lpfnDlg);
```

Declaring Functions Within Your Application

Make sure all application functions are declared. Placement of all function declarations in an include file is highly recommended because you can more easily scan through your function declarations and look for parameter and return types that should be changed.

If you use the **/Zg** compiler option to create header files for your functions, remember that you'll get different results depending on whether or not you have enabled STRICT type checking. With STRICT disabled, all handle types generate the same base type: **unsigned short**. With STRICT enabled, they generate base types such as **HWND __near *** or **HDC __near ***. To avoid conflict, you need to either recreate the header file each time you disable or enable STRICT, or else edit the header file to use the types **HWND**, **HDC**, **HANDLE**, and so on, instead of the base types.

If you've copied any function declarations from WINDOWS.H into your source code, they may have changed, and your local declaration may be out of date. Remove your local declaration.

Functions That Require Casts

Some functions have generic return types or parameters. For example, a function like **SendMessage** returns data that may be any number of types, depending on the context. When you see any of these functions in your source code, make sure that you use the correct type cast and that it is as specific as possible.

The following table summarizes these functions:

Function	Comment
LocalLock	Cast result to the proper kind of data pointer.
GlobalLock	Cast result to the proper kind of data pointer.
GetWindowWord	Cast result to appropriate data type.
GetWindowLong	Cast result to appropriate data type
SetWindowWord	Cast argument as it is passed to function.
SetWindowLong	Cast argument as it is passed to function.
SendMessage	Cast result to appropriate data type; cast to **UINT** before casting to a handle type.
DefWindowProc	See comment for **SendMessage**.
SendDlgItemMessage	See comment for **SendMessage**.

When you call **SendMessage**, **DefWindowProc**, or **SendDlgItemMessage**, you should first cast the result to type **UINT**. You need to take similar steps for any function that returns **LRESULT** or **LONG**, where the result contains a handle. This is necessary for writing portable code because the size of a handle is either 16 bits or 32 bits depending on the version of Windows. The **(UINT)** cast ensures proper conversion. The following code shows an example in which **SendMessage** returns a handle to a brush:

```
HBRUSH hbr;

hbr = (HBRUSH)(UINT)SendMessage(hwnd, WM_CTLCOLOR, ..., ...);
```

The CreateWindow Function

The **CreateWindow** and **CreateWindowEx** *hmenu* parameter is sometimes used to pass an integer control ID. In this case, you must cast this to an **HMENU** type:

```
HWND hwnd;
int id;

hwnd = CreateWindow("Button", "Ok", BS_PUSHBUTTON,
        x, y, cx, cy, hwndParent,
        (HMENU)id,      // Cast required here
        hinst,
        NULL);
```

Making Best Use of STRICT Type Checking

To get the most benefit from STRICT type checking, there are other guidelines you should follow in addition to those in Making Your Application STRICT Compliant. Your code will be more portable in future versions of Windows if you make the following changes:

Change	To
HANDLE	A specific handle such as **HINSTANCE**, **HMODULE**, **HGLOBAL**, **HLOCAL**, and so on
WORD	**UINT**, except where you want a 16-bit value even when the platform is 32 bits
WORD	**WPARAM**, where *wParam* is declared
LONG	**LPARAM** or **LRESULT** as appropriate

Any time you need an integer data type, you should declare it as **UINT** except where a 16-bit value is specifically required (as in a structure or parameter). For even if a variable never exceeds the range of a 16-bit integer, it can be more efficiently handled by the processor if it is 32 bits.

The types **WPARAM**, **LPARAM**, **LRESULT**, and **void *** are "polymorphic data types": they hold different kinds of data at different times, even when STRICT type checking is enabled. To get the benefit of type checking, you should cast values of these types as soon as possible. Note that message crackers (as well as the Microsoft Foundation Classes) automatically recast *wParam* and *lParam* for you in a portable way.

Take special care to distinguish **HMODULE** and **HINSTANCE** types. Even with STRICT enabled, they are defined as the same base type. Most kernel module management functions use **HINSTANCE** types, but there are a few functions that return or accept only **HMODULE** types.

Accessing the New COMSTAT Structure

The Windows 3.0 declaration of the **COMSTAT** structure is not compatible with ANSI standards. WINDOWS.H now defines the **COMSTAT** structure to be compatible with ANSI compilers and so that the **/W4** option does not issue warnings.

To support backward compatibility of source code, WINDOWS.H does not use the new structure definition unless the Windows version (as indicated by WINVER) is greater than 3.0 or if STRICT is defined. When you enable STRICT, the presumption is that you are trying to write portable code. Therefore, WINDOWS.H uses the new **COMSTAT** structure for all versions of Windows if STRICT is enabled.

The new structure definition replaces the bit fields by flags accessing bits in a single field, named **status**, as shown below. Each flag turns on a different bit.

Windows 3.0 field name	Flag accessing the status field
fCtsHold	CSTF_CTSHOLD
fDsrHold	CSTF_DSRHOLD
fEof	CSTF_EOF
fRlsdHold	CSTF_RLSDHOLD
fTxim	CSTF_TXIM
fXoffHold	CSTF_XOFFHOLD
fXoffSent	CSTF_XOFFSENT

If you have code that accesses any of these status fields, you need to change your code as appropriate. For example, suppose you have the following code written for Windows 3.0:

```
if (comstat.fEof || fCondition)
     comstat.fCtsHold = TRUE;
     comstat.fTxim = FALSE;
```

This code should be replaced by code that access individual bits of the **status** field by using flags. Note the use of bitwise operators.

```
if ((comstat.status & CSTF_EOF) || fCondition)
     comstat.status |= CSTF_CTSHOLD;
     comstat.status ~= CSTF_TXIM;
```

Interpreting Error Messages Affected by STRICT

Enabling STRICT type checking may affect the kind of error messages you receive. With STRICT enabled, all handle types as well as the types **LRESULT**, **WPARAM**, and **LPARAM** are defined as pointer types. When you incorrectly use these types (for example, passing an **int** where an **HDC** is expected), you will get error messages referring to errors in pointer indirection.

Another effect of STRICT is to require that **FARPROC** function pointers be recast as more specific function pointer types such as **DLGPROC**. However, **MakeProcInstance** and **FreeProcInstance** still work with the **FARPROC** type. If you fail to cast between **FARPROC** and the appropriate function pointer type, the compiler will report an error in function parameter lists.

Note that using **MakeProcInstance** is useful for the sake of portability, if you want to use the same source to compile for Windows 3.*x*. Under Win32, however, **MakeProcInstance** performs no operation, but just returns the function name.

Storage Class Attributes

This section describes extended attribute syntax, which simplifies and standardizes the Microsoft-specific extensions to the Microsoft C and C++ languages. The storage class attributes that use extended attribute syntax include **thread**, **naked**, **dllimport**, and **dllexport**.

Extended Attribute Syntax (Microsoft Specific)

The extended attribute syntax for specifying storage class information uses the **declspec** keyword, which specifies that an instance of a given type is to be stored with a Microsoft-specific storage class attribute (**thread**, **naked**, **dllimport**, or **dllexport**). Some examples of other storage class modifiers include the **static** and **extern** keywords. However, these keywords are part of the ANSI specification of the C and C++ languages, and as such are not covered by extended attribute syntax.

This is the extended attribute syntax for C:

Syntax

storage-class-specifier:
- **typedef**
- **extern**
- **static**
- **auto**
- **register**
- **__declspec** (*extended-decl-modifier-list*)

For C++, the syntax looks like this:

Syntax

decl-specifier:
- *storage-class-specifier*
- *type-specifier*
- *fct-specifier*
- **friend**
- **typedef**
- **__declspec** (*extended-decl-modifier-list*)

For C and C++, the *extended-decl-modifier-list* syntax looks like this:

Syntax

extended-decl-modifier-list:
- *extended-decl-modifier*
- *extended-decl-modifier extended-decl-modifier-list*

extended-decl-modifier:
- **thread**
- **naked**
- **dllimport**
- **dllexport**

White space separates the declaration modifier lists. Examples of the syntax appear in later sections.

The **thread**, **naked**, **dllimport**, and **dllexport** storage class attributes are a property only of the declaration of the object or function to which they are applied. Unlike the **__near** and **__far** keywords, which actually affect the type of object or function (in this case, 2- and 4-byte addresses), these storage class attributes do not redefine the type attributes of the object itself. The **thread** attribute affects data and objects only. The **naked** attribute affects functions only. The **dllimport** and **dllexport** attributes affect functions, data, and objects.

The following statement declares an integer variable, **tls_i**, which is used to store thread-specific data.

```
__declspec( thread ) int tls_i = 1;
```

To make your code more readable, you can control such declarations using macro definitions:

```
#define Thread  __declspec( thread )
Thread int tls_i = 1;
```

The Thread Attribute (32-bit Specific)

Thread Local Storage (TLS) is the mechanism by which each thread in a given multi-threaded process allocates storage for thread-specific data. In standard multi-threaded programs, data is shared among all threads of a given process, whereas thread-local storage is the mechanism for allocating per-thread data. For a complete discussion of threads, see Processes and Threads.

The C and C++ languages include a new extended storage class attribute, **thread**. The **thread** attribute must be used with the **__declspec** keyword to declare a thread variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

Rules and Limitations

The following guidelines must be observed when you are declaring statically bound thread local objects and variables.

- You can apply the **thread** attribute only to data declarations and definitions. It cannot be used on function declarations or definitions. For example, the following code generates a compiler error:

  ```
  #define Thread __declspec( thread )
  Thread void func();        // Error
  ```

- You can specify the **thread** attribute only on data items with static storage duration. This includes global data objects (both **static** and **extern**), local static objects, and static data members of C++ classes. You cannot declare automatic data objects with the **thread** attribute. For example, the following code generates compiler errors:

  ```
  #define Thread __declspec( thread )
  void func1()
  {
     Thread int tls_i;                  // Error
  }

  int func2( Thread int tls_i )         // Error
  {
     return tls_i;
  }
  ```

- You must use the **thread** attribute for the declaration and the definition of a thread local object, regardless of whether the declaration and definition occur in the same file or separate files. For example, the following code generates an error:

  ```
  #define Thread __declspec( thread )
  extern int tls_i;          // This generates an error, because the
  int Thread tls_i;          // declaration and the definition differ.
  ```

- You cannot use the **thread** attribute as a type modifier. For example, the following code generates a compiler error:

  ```
  char __declspec( thread ) *ch;         // Error
  ```

- C++ classes cannot use the **thread** attribute. However, you can instantiate C++ class objects with the **thread** attribute. For example, the following code generates a compiler error:

  ```
  #define Thread __declspec( thread )
  class Thread C       // Error: classes can't be declared Thread.
  {
  ```

```
// Code
};
C CObject;
```

Because the declaration of C++ objects that use the **thread** attribute is permitted, these two examples are semantically equivalent:

```
#define Thread __declspec( thread )
Thread class B
{
// Code
} BObject;                        // Okay--BObject declared thread local.

class B
{
// Code
}
Thread B BObject;          // Okay--BObject declared thread local.
```

- Because C++ objects with constructors and destructors, as well as objects that use initialization semantics, can be allocated as thread local, an associated initialization routine must be called to initialize the object. In this example, the constructor initializes the thread local object:

```
class tlsClass
{
   private:
         int x;
   public:
         tlsClass() { x = 1; } ;
         ~tlsClass();
}

__declspec( thread ) tlsClass tlsObject;
extern int func();
__declspec( thread ) int y = func();
```

- The address of a thread local object is not considered constant, and any expression involving such an address is not considered a constant expression. In C and C++, this means that you cannot use the address of a thread local variable as an initializer for an object or pointer. For example, the compiler flags the following code as an error:

```
#define Thread __declspec( thread )
Thread int tls_i;
int *p = &tls_i;           //Error
```

- Standard C permits initialization of an object or variable with an expression involving a reference to itself, but only for objects of non-static extent. Although C++ normally permits such dynamic initialization of an object with an expression involving a reference to itself, this type of initialization is not permitted with thread local objects. For example:

```
#define Thread __declspec( thread )
Thread int tls_i = tls_i;                          // C and C++ error
int j = j;                                                  // Okay in
C++; C error
Thread int tls_i = sizeof( tls_i )                 // Okay in C and C++
```

Note that a **sizeof** expression that includes the object being initialized does not constitute a reference to itself, and is allowed in C and C++.

The Naked Attribute (32-bit Specific)

For functions declared with the **naked** attribute, the compiler generates code without prolog and epilog code. You can use this feature to write your own prolog/epilog code sequences using inline assembly code. Naked functions are particularly useful in writing virtual device drivers and interrupt handlers.

Because the **naked** attribute is only relevant to the definition of a function and is not a type modifier, naked functions use the extended attribute syntax, described previously. For example, this code defines a function with the **naked** attribute:

```
__declspec( naked ) int func( formal_parameters )
{
     // Function body
}
```

Or, alternatively:

```
#define Naked    __declspec( naked )
Naked int func( formal_parameters )
{
     // Function body
}
```

The **naked** modifier affects only the nature of the compiler's code generation for the function's prolog and epilog sequences. It does not affect the code that is generated for calling such functions. Thus, the **naked** attribute is not considered part of the function's type, and function pointers cannot have the **naked** attribute. Furthermore, the **naked** attribute cannot be applied to a data definition. For example, the following code samples generate errors:

```
__declspec( naked ) int i;          // Error--naked attribute not
                                              // permitted on data
                                              // declarations.
```

The **naked** attribute is relevant only to the definition of the function and cannot be specified in the function's prototype. The following declaration generates a compiler error:

```
__declspec( naked ) int func();           // Error--naked attribute
                                                    // not permitted on
function
                                                    // declarations.
```

Rules and Limitations

- The **return** statement is not permitted in a naked function. However, you can return an **int** by moving the return value into the eax register before the ret instruction.
- Structured exception handling constructs are not permitted in a naked function because the constructs must unwind across the stack frame.
- Any use of the **setjmp** run-time function is not permitted in a naked function because it too must unwind across the stack frame. However, use of the **longjmp** run-time function is permitted.
- Use of the **_alloca** function is not permitted in a naked function.
- To ensure that no initialization code for local variables appears before the prolog sequence, initialized local variables are not permitted at function scope. In particular, the declaration of C++ objects is not permitted at function scope. There can, however, be initialized data in a nested scope.
- Frame pointer optimization (the **/Oy** compiler option) is not recommended, but it is automatically suppressed for a naked function.

Considerations for Writing Prolog/Epilog Code

Before writing your own prolog and epilog code sequences, it is important to understand how the stack frame is laid out, and some differences in stack frame layout between 16-bit and 32-bit targets. It is also useful to know how to use the **_LOCAL_SIZE** symbol. These topics are discussed in the following sections.

Stack Frame Layout

A few minor differences exist between 16- and 32-bit stack frame layout. This example shows the standard prolog code that might appear in a 32-bit function:

```
push       ebp                          ; Save ebp
mov              ebp, esp                     ; Set stack frame pointer
sub              esp, localbytes          ; Allocate space for locals
push       registers                ; Save registers
```

The *localbytes*variable represents the number of bytes needed on the stack for local variables, and the *registers* variable is a placeholder that represents the list of registers to be saved on the stack. After pushing the registers, you can place any other appropriate data on the stack. The following is the corresponding epilog code:

```
pop              registers                ; Restore registers
mov              esp, ebp                      ; Restore stack pointer
pop              ebp                           ; Restore ebp
ret                                            ; Return from function
```

The stack always grows down (from high to low memory addresses). The base pointer (*ebp*) points to the pushed value of *ebp*. The locals area begins at *ebp-2*. To access local variables, calculate an offset from ebp by subtracting the appropriate value from *ebp*.

The only change in these code sequences for 16-bit targets is that the register names do not begin with an *e*. For 16-bit targets, the register names in the examples above would be **bp** and **sp**.

__LOCAL_SIZE

The compiler provides a symbol, **__LOCAL_SIZE**, for use in the inline assembly block of function prolog code. This symbol is used to allocate space for local variables on the stack frame in custom prolog code.

The compiler determines the value of **__LOCAL_SIZE**. Its value is the total number of bytes of all user-defined local variables and compiler-generated temporary variables. **__LOCAL_SIZE** can be used only as an immediate operand; it cannot be used in an expression. You must not change or redefine the value of this symbol. For example:

```
mov         eax, __LOCAL_SIZE                   ;Immediate operand--Okay
mov         eax, __LOCAL_SIZE + 4               ;Error
mov         eax, [ebp - __LOCAL_SIZE]           ;Error
```

The following example of a naked function containing custom prolog and epilog sequences uses **__LOCAL_SIZE** symbol in the prolog sequence:

```
__declspec ( naked ) func()
{
     int i;
     int j;

     __asm       /* prolog */
           {
           push  ebp
           mov         ebp, esp
           sub         esp, __LOCAL_SIZE
           }
     /* Function body */

     __asm       /* epilog */
           {
           mov         esp, ebp
           pop         ebp
           ret
           }
}
```

The Dllexport and Dllimport Attributes (32-bit Specific)

The **dllexport** and **dllimport** storage class modifiers export and import functions, data, and objects to and from a DLL. These modifiers, or attributes, explicitly define the DLL's interface to its client, which can be the executable file or another DLL. Declaring functions as **dllexport** eliminates the need for a module-definition (.DEF) file, at least with respect to the specification of exported functions. Note that **dllexport** replaces the **__export** keyword.

The declaration of **dllexport** and **dllimport** uses extended attribute syntax:

```
#define DllImport__declspec( dllimport )
#define DllExport__declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

Definitions and Declarations

The DLL interface refers to all items (functions and data) that are known to be exported by some program in the system, that is, all items that are declared as **dllimport** or **dllexport**. All declarations included in the DLL interface must specify either the **dllimport** or **dllexport** attribute. However, the definition can specify only the **dllexport** attribute. For example, the following function definition generates a compiler error:

```
#define DllImport__declspec( dllimport )
#define DllExport__declspec( dllexport )

DllImport int func()         // Error; dllimport prohibited on definition.
{
     return 1;
}
```

This code also generates an error:

```
DllImport int i = 10;        // Error; this is a definition.
```

However, this is correct syntax:

```
DllExport int i = 10;        // Okay; this is an export definition.
```

The use of **dllexport** implies a definition, while **dllimport** implies a declaration. You must use the **extern** keyword with **dllexport** to force a declaration; otherwise, a definition is implied. Thus, the following examples are correct:

```
extern DllImport int k;      // These are both correct and imply a
DllImport int j;             // declaration.
```

The following examples help to further clarify the above:

```
static DllImport int l;            // Error; not declared extern.

void func()
{
     static DllImport int s;       // Error; not declared extern.
     DllImport int m;              // Okay; this is a declaration.
     DllExport int n;              // Error; implies external
                                              // definition in local scope.

     extern DllImport int i;       // Okay; this is a declaration.
     extern DllExport int k;       // Okay; extern implies
                                              // declaration.

     DllExport int x = 5;          // Error; implies external
                                              // definition in local scope.
}
```

Defining Inline Functions with Dllexport and Dllimport

You can define as inline a function with the **dllexport** attribute. In this case, the function is always instantiated and exported, whether or not any module in the program references the function. The function is presumed to be imported by another program.

You can also define as inline a function declared with the **dllimport** attribute. In this case, the function can be expanded (subject to **/Ob** compiler switch specifications), but never instantiated. In particular, if the address of an inline imported function is taken, the address of the function residing in the DLL is returned. This behavior is the same as taking the address of a non-inline imported function.

These rules apply to inline functions whose definitions appear within a class definition. In addition, static local data and strings in inline functions maintain the same identities between the DLL and client as they would in a single program (that is, an executable file without a DLL interface).

Exercise care when providing imported inline functions. For example, if you update the DLL, don't assume that the client will use the changed version of the DLL. To ensure that you are loading the proper version of the DLL, rebuild the DLL's client as well.

General Rules and Limitations

- If you declare a function or object without the **dllimport** or **dllexport** attribute, the function or object is not considered part of the DLL interface. Therefore, the definition of the function or object must be present in that module or in another module of the same program. To make the function or object part of the DLL interface, you must declare the definition of the function or object in the other module as **dllexport**. Otherwise, a linker error is generated.

  If you declare a function or object with the **dllexport** attribute, its definition must appear in some module of the same program. Otherwise, a linker error is generated.
- If a single module in your program contains both **dllimport** and **dllexport** declarations for the same function or object, the **dllexport** attribute takes precedence over the **dllimport** attribute. However, a compiler warning is generated. For example:

```
DllImport int i;
DllExport int i;        // Warning; inconsistent, but
                                    // dllexport takes precedence.
```

- In C, if you initialize a globally declared pointer with the address of a data object declared with the **dllimport** attribute, a compiler error is generated. Similarly, you cannot initialize a static local function pointer with the address of a function declared with the **dllimport** attribute, or initialize a static local data pointer with the address of a data object declared with the **dllimport** attribute. The C++ compiler does not enforce this restriction, because C++ supports dynamic initialization of local and global static objects. For example, the following code generates errors when compiled with the C compiler, but not with the C++ compiler:

```
DllImport void func1( void );
DllImport int i;

int *pi = &i;                                                    //
Error in C
static void ( *pf )( void ) = &func1;                 // Error in C
void func2()
{
      static int *pi = &i;                                    // Error in
C
      static void ( *pf )( void ) = &func1;           // Error in C
}
```

  However, because a program that includes the **dllexport** attribute in the declaration of an object must provide the definition for that object somewhere in the program, you can initialize a global or local static function pointer with the address of a **dllexport** function. Similarly, you can initialize a global or local static data pointer with the address of a **dllexport** data object. For example, the following code does not generate errors in C or C++:

```
DllExport void func1( void );
DllExport int i;

int *pi = &i;                                                    //
Okay
static void ( *pf )( void ) = &func1;                 // Okay

void func2()
{
      static int *pi = &i;                                    // Okay
      static void ( *pf )( void ) = &func1;           // Okay
```

```
}
```

C++ Specific Rules and Limitations

You can declare C++ classes with the **dllimport** or **dllexport** attribute. These forms imply that the entire class is imported or exported. Classes exported in this manner are referred to as exportable classes.

The following example defines an exportable class. All its member functions and static data are exported:

```
class DllExport C
{
     int i;
     virtual int func( void )
     { return 1; }
};
```

Note that explicit use of the **dllimport** and **dllexport** attributes on members of an exportable class is prohibited.

Dllexport Classes

When you declare a class **dllexport**, all its member functions and static data members are exported. You must provide the definitions of all such members in the same program. Otherwise, a linker error is generated. The one exception to this rule applies to pure virtual functions, for which you need not provide explicit definitions. However, because a destructor for an abstract class is always called by the destructor for the base class, pure virtual destructors must always provide a definition. Note that these rules are the same for non-exportable classes.

If you export data of class type or functions that return classes, be sure to export the class.

Dllimport Classes

When you declare a class **dllimport**, all its member functions and static data members are imported. Unlike the behavior of **dllimport** and **dllexport** on non-class types, static data members cannot specify a definition in the same program in which a **dllimport** class is defined.

Inheritance and Exportable Classes

All base classes of an exportable class must be exportable. If not, a compiler warning is generated. Moreover, all accessible members that are also classes must be exportable. This rule permits a **dllexport** class to inherit from a **dllimport** class, and a **dllimport** class to inherit from a **dllexport** class (though the latter is not recommended). As a rule, everything that is accessible to the DLL's client (according to C++ access rules) should be part of the exportable interface. This includes private data members referenced in inline functions.

Selective Member Import/Export

Because member functions and static data within a class implicitly have external linkage, you can declare them with the **dllimport** or **dllexport** attributes, unless the entire class is exported. If the entire class is imported or exported, the explicit declaration of member functions and data as **dllimport** or **dllexport** is prohibited. If you declare a static data member within a class definition as **dllexport**, a definition must occur somewhere within the same program (as with non-class external linkage).

Similarly, you can declare member functions with the **dllimport** or **dllexport** attributes. In this case, you must provide a **dllexport** definition somewhere within the same program.

It is worthwhile to note several important points regarding selective member import and export:

- Selective member import/export is best used for providing a version of the exported class interface that is more restrictive; that is, one for which you can design a DLL that exposes fewer public and private features than the language would otherwise allow. It is also useful for fine tuning the exportable interface: when you know that the client, by definition, is unable to access some private data, you need not export the entire class.
- If you export one virtual function in a class, you must export all of them, or at least provide versions that the client can use directly.

  If you have a class in which you are using selective member import/export with virtual functions, the functions must be in the exportable interface or defined inline (visible to the client).
- If you define a member as **dllexport** but do not include it in the class definition, a compiler error is generated. You must define the member in the class header.
- Although the definition of class members as **dllimport** or **dllexport** is permitted, you cannot override the interface specified in the class definition.
- If you define a member function in a place other than the body of the class definition in which you declared it, a warning is generated if the function is defined as **dllexport** or **dllimport** (if this definition differs from that specified in the class declaration).

Shared Header Files

A good way to ensure that the DLL and its client agree on the interface to a DLL is to use the same header files for building the DLL and the client. In this case, all header files should use **dllimport** since **dllexport** overrides **dllimport** in the case of definition.

Legal Notice

Olivetti is a registered trademark of Ing. C. Olivetti.

PostScript is a registered trademark of Adobe Systems, Inc.

R4000 is a trademark of MIPS Computer Systems, Inc.

Roland is a registered trademark of Roland Corporation.

SCSI is a registered trademark of Security Control Systems, Inc.

Epson is a registered trademark of Seiko Epson Corporation, Inc.

Silicon Graphics is a registered trademark and OpenGL is a trademark of Silicon Graphics, Inc.

Stacker is a registered trademark of STAC Electronics.

Tandy is a registered trademark of Tandy Corporation.

Unicode is a registered trademark of Unicode, Incorporated.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

VAX is a trademark of Digital Equipment Corporation

Yamaha is a registered trademark of Yamaha Corporation of America.

Paintbrush is a trademark of Wordstar Atlanta Technology Center.

Microsoft Win32 Developer's Reference

You have requested information from the **Microsoft Win32 Developer's Reference**. One or more of these help files is not available on your system.