CHAPTER 10

Writing a Dynamic-Link Library For Windows

The Windows operating system relies heavily on service routines and data contained in special libraries called "dynamic-link libraries," or DLLs for short. Most of what Windows comprises, from the collections of screen fonts to the routines that handle the graphical interface, is provided by DLLs. MASM 6.1 contains tools that you can use to write DLLs in assembly language. This chapter shows you how.

DLLs do not run under MS-DOS. The information in this chapter applies only to Windows, drawing in part on the chapter "Writing a Module-Definition File" in *Environment and Tools*. The acronym API, which appears throughout this chapter, refers to the application programming interface that Windows provides for programs. For documentation of API functions, see the *Programmer's Reference, Volume 2* of the Windows Software Development Kit (SDK).

The first section of this chapter gives an overview of DLLs and their similarities to normal libraries. The next section explores the parts of a DLL and the rules you must follow to create one. The third section applies this information to an example DLL.

Overview of DLLs

A dynamic-link library is similar to a normal run-time library. Both types of libraries contain a collection of compiled procedures, which serve one or more calling modules. To link a normal library, the linker copies the required functions from the library file (which usually has a .LIB extension) and combines them with other modules to form an executable program in .EXE format. This process is called static linking.

In dynamic linking, the library functions are not copied to an .EXE file. Instead, they reside in a separate file in executable form, ready to serve any calling program, called a "client." When the first client requires the library, Windows takes care of loading the functions into memory and establishing linkage. If

subsequent clients also need the library, Windows dynamically links them with the proper library functions already in memory.

Loading a DLL

How Windows loads a DLL affects the client rather than the DLL itself. Accordingly, this section focuses on how to set up a client program to use a DLL. Since the client can itself be a DLL, this is information a DLL programmer should know. However, MASM 6.1 does not provide all the tools required to create a stand-alone program for Windows. To create such a program, called an "application," you must use tools in the Windows SDK.

Windows provides two methods for loading a dynamic-link library into memory:

Method	Description
Implicit loading	Windows loads the DLL along with the first client program and links it before the client begins execution.
Explicit loading	Windows does not load the DLL until the first client explicitly requests it during execution.

When you write a DLL, you do not need to know beforehand which of the two methods will be used to load the library. The loading method is determined by how the client is written, not the DLL.

Implicit Loading

The implicit method of loading a DLL offers the advantage of simplicity. The client requires no extra programming effort and can call the library functions as if they were normal run-time functions. However, implicit loading carries two constraints:

- The name of the library file must have a .DLL extension.
- You must either list all DLL functions the client calls in the IMPORTS section of the client's module-definition file, or link the client with an import library.

An import library contains no executable code. It consists of only the names and locations of exported functions in a DLL. The linker uses the locations in the import library to resolve references to DLL functions in the client and to build an executable header. For example, the file LIBW.LIB provided with MASM 6.1 is the import library for the DLL files that contain the Windows API functions.

The IMPLIB utility described in *Environment and Tools* creates an import library. Run IMPLIB from the MS-DOS command line like this:

IMPLIB *implibfile dllfile*

where *implibfile* is the name of the import library you want to create from the DLL file *dllfile*. Once you have created an import library from a DLL, link it with a client program that relies on implicit loading, but does not list imported functions in its module-definition file. Continuing the preceding example, here's the link step for a client program that calls library procedures in the DLL *dllfile*:

LINK *client*.OBJ, *client*.EXE, , *implibfile*, *client*.DEF

This simplified example creates the client program *client*.EXE, linking it with the import library *implibfile*, which in turn was created from the DLL file *dllfile*.

To summarize implicit loading, a client program must either

- List DLL functions in the IMPORTS section of its module-definition file, or
- Link with an import library created from the DLL.

Implicit loading is best when a client always requires at least one procedure in the library, since Windows automatically loads the library with the client. If the client does not always require the library service, or if the client must choose at run time between several libraries, you should use explicit loading, discussed next.

Explicit Loading

To explicitly load a DLL, the client does not require linking with an import library, nor must the DLL file have an extension of .DLL. Explicit loading involves three steps in which the client calls Windows API functions:

1. The client calls **LoadLibrary** to load the DLL.
2. The client calls **GetProcAddress** to obtain the address of each DLL function it requires.
3. When finished with the DLL, the client calls **FreeLibrary** to unload the DLL from memory.

The following example fragment shows how a client written in assembly language explicitly loads a DLL called SYSINFO.DLL and calls the DLL function **`GetSysDate`**.

```
        INCLUDE windows.inc
        .DATA
hInstance       HINSTANCE 0
szDLL           BYTE    'SYSINFO.DLL', 0
szDate          BYTE    'GetSysDate', 0
lpProc          DWORD   0
```

```
.CODE
.
.
.
INVOKE  LoadLibrary, ADDR szDLL         ; Load SYSINFO.DLL
mov     hInstance, ax                   ; Save instance count
INVOKE  GetProcAddress, ax, ADDR szDate ; Get and save
mov     lpProc, ax                      ;   far address of
mov     lpProc[2], dx                   ;   GetSysDate
call    lpProc                          ; Call GetSysDate
.
.
.
INVOKE  FreeLibrary, hInstance  ; Unload SYSINFO.DLL
```

For simplicity, the above example contains no error-checking code. An actual program should check all values returned from the API functions.

The explicit method of loading a DLL requires more programming effort in the client program. However, the method allows the client to control which (if any) dynamic-link libraries to load at run time.

Searching for a DLL File

To load a DLL, whether implicitly or explicitly, Windows searches for the DLL file in the following directories in the order shown:

1. The current directory
2. The Windows directory, which contains WIN.COM
3. The Windows system directory, which contains system files such as GDI.EXE
4. The directory where the client program resides (except Windows 3.0 and earlier)
5. Directories listed in the PATH environment string
6. Directories mapped in a network

If Windows does not locate the DLL in any of these directories, it prompts the user with a message box.

Building a DLL

A DLL has additional programming requirements beyond those for a normal run-time library. This section describes the requirements pertaining to the library's code, data, and stack. It also discusses the effects of the library's extension name.

DLL Code

The code in a DLL consists of exported and nonexported functions. Exported functions, listed in the **EXPORTS** section of the module-definition file, are public routines serving clients. Nonexported functions provide private, internal support for the exported procedures. They are not visible to a client.

Under Windows, an exported library routine must appear to the caller as a far procedure. Your DLL routines can use any calling convention you wish, provided the caller assumes the same convention. You can think of dynamic-link code as code for a normal run-time library with the following additions:

- An entry procedure
- A termination procedure
- Special prologue and epilogue code

Entry Procedure

A DLL, like any Windows-based program, must have an entry procedure. Windows calls the entry procedure only once when it first loads the DLL, passing the following information in registers:

- DS contains the library's data segment address.
- DI holds the library's instance handle.
- CX holds the library's heap size in bytes.

Note Windows API functions destroy all registers except DI, SI, BP, DS, and the stack pointer. To preserve the contents of other registers, your program must save the registers before an API call and restore them afterwards.

This information corresponds to the data provided to an application. Since a DLL has only one occurrence in memory, called an "instance," the value in DI is not usually important. However, a DLL can use its instance handle to obtain resources from its own executable file.

The entry procedure does not need to record the address of the data segment. Windows automatically ensures that each exported routine in the DLL has access to the library's data segment, as explained in "Prologue and Epilogue Code," on page 264.

The heap size contained in CX reflects the value provided in the **HEAPSIZE** statement of the module-definition file. You need not make an accurate guess in the **HEAPSIZE** statement about the library's heap requirements, provided you specify a moveable data segment. With a moveable segment, Windows automatically allocates more heap when needed. However, Windows can

provide no more heap in a fixed data segment than the amount specified in the **HEAPSIZE** statement. In any case, a library's total heap cannot exceed 64K, less the amount of static data. Static data and heap reside in the same segment.

Windows does not automatically deallocate unneeded heap while the DLL is in memory. Therefore, you should not set an unnecessarily large value in the **HEAPSIZE** statement, since doing so wastes memory.

The entry procedure calls the Windows API function **LocalInit** to allocate the heap. The library must create a heap before its routines call any heap functions, such as **LocalAlloc**. The following example illustrates these steps:

```
DLLEntry PROC FAR PASCAL PUBLIC         ; Entry point for DLL

        jcxz    @F                      ; If no heap, skip
        INVOKE  LocalInit, ds, 0, cx    ; Else set up the heap
        .IF     ( ax )                  ; If successful,
        INVOKE  UnlockSegment, -1       ;   unlock the data segment
@@:     call    LibMain                 ; Call DLL's data init routine
        mov     ax, TRUE                ; Return AX = 1 if okay,
        .ENDIF                          ;   else if LocalInit error,
        ret                             ;   return AX = 0

DLLEntry ENDP
```

This example code is taken from the DLLENTRY.ASM module, contained in the LIB subdirectory on one of the MASM 6.1 distribution disks. After allocating the heap, the procedure calls the library's initialization procedure—called **LibMain** in this case. **LibMain** initializes the library's static data (if required), then returns to **DLLEntry**, which returns to Windows. If Windows receives a return value of 0 (FALSE) from **DLLEntry**, it unloads the library and displays an error message.

The process is similar to the way MS-DOS loads a terminate-and-stay-resident program (TSR), described in the next chapter. Both the DLL and TSR return control immediately to the operating system, then wait passively in memory to be called.

The following section explains how a DLL gains control when Windows unloads it from memory.

Termination Procedure

Windows maintains a DLL in memory until the last client program terminates or explicitly unloads the library. When unloading a DLL, Windows first calls the library's termination procedure. This allows the DLL to return resources and do any necessary cleanup operations before Windows unloads the library from memory.

Libraries that have registered window procedures with **RegisterClass** need not call **UnregisterClass** to remove the class registration. Windows does this automatically when it unloads the library.

You must name the library's termination procedure `WEP` (for Windows Exit Procedure) and list it in the **EXPORTS** section of the library's module-definition file. To ensure immediate operation, provide an ordinal number and use the **RESIDENTNAME** keyword, as described in the chapter "Creating Module-Definition Files" in *Environment and Tools.* This keeps the name "WEP" in the Windows-resident name table at all times.

Besides its name, the code for `WEP` should also remain constantly in memory. To ensure this, place `WEP` in its own code segment and set the segment's attributes as **PRELOAD FIXED** in the **SEGMENTS** statement of the module-definition file. Thus, your DLL code should use a memory model that allows multiple code segments, such as medium model. Since a termination procedure is usually short, keeping it resident in memory does not burden the operating system.

The termination procedure accepts a single parameter, which can have one of two values. These values are assigned to the following symbolic constants in the WINDOWS.INC file located in the LIB subdirectory:

- `WEP_SYSTEM_EXIT` (value 1) indicates Windows is shutting down.
- `WEP_FREE_DLL` (value 0) indicates the library's last client has terminated or has called **FreeLibrary**, and Windows is unloading the DLL.

The following fragment provides an outline for a typical termination procedure:

```
WEP     PROC FAR PASCAL EXPORT
        wExitCode:WORD

        Prolog                              ; Prologue macro,
        .IF     wExitCode == WEP_FREE_DLL   ;   discussed below
        .                                   ; Get ready to
        .                                   ;   unload
        .
        ELSEIF  wExitCode == WEP_SYSTEM_EXIT
        .                                   ; Windows is
        .                                   ;   shutting down
        .
        . ENDIF                             ; If neither value,
                                            ;   take no action
        mov     ax, TRUE                    ; Always return AX = 1
        Epilog                              ; Epilogue code,
        ret                                 ;   discussed below

WEP     ENDP
```

Usually, the WEP procedure takes the same actions regardless of the parameter value, since in either case Windows will unload the DLL.

Under Windows 3.0, the WEP procedure receives stack space of about 256 bytes. This allows the procedure to unhook interrupts, but little else. Any other action, such as calling an API function, usually results in an unrecoverable application error because of stack overflow. Later versions of Windows provide at least 4K of stack to the WEP procedure, allowing it to call many API functions.

However, WEP should not send or post a message to a client, because the client may already be terminated. The WEP procedure should also not attempt file I/O, since only application processes—not DLLs—can own files. When control reaches WEP, the client may no longer exist and its files are closed.

Prologue and Epilogue Code

Exported procedures in a Windows-based program require special epilogue and prologue code. (For a definition of these terms, see "Generating Prologue and Epilogue Code" in Chapter 7.) The SAMPLES subdirectory on one of the MASM 6.1 distribution disks contains macros you can use for far procedures in your Windows-based programs. Here's a listing of the prologue macro:

```
Prolog  MACRO
        mov     ax, ds          ; Must be 1st, since Windows overwrites
        nop                     ; Placeholder for 3rd byte
        inc     bp              ; Push odd BP.  Not required, but
        push    bp              ;   allows CodeView to recognize frame
        mov     bp, sp          ; Set up stack frame to access params
        push    ds              ; Save DS
        mov     ds, ax          ; Point DS to DLL's data segment
        ENDM
```

The instruction

```
        inc     bp
```

marks the beginning of the stack frame with an odd number. This allows real-mode Windows to locate segment addresses on the stack and update the addresses when it moves or discards the corresponding segments. In protected mode, selector values do not change when segments are moved, so marking the stack frame is not required. However, certain debugging applications, such as Microsoft Codeview for Windows and the Microsoft Windows 80386 Debugger (both documented in *Programming Tools* of the SDK), search for a marked frame to determine if the frame belongs to a far procedure. Without the mark, these debuggers give meaningless information when backtracing through the stack. Therefore, you should include the **INC BP** instruction for Windows-

based programs that may run in real mode or that require debugging with a Microsoft debugger.

Another characteristic of the prologue macro may seem puzzling at first glance. The macro moves DS into AX, then AX back into DS. This sequence of instructions lets Windows selectively overwrite the prologue code in far procedures. When Windows loads a program, it compares the names of far procedures with the list of exported procedures in the module-definition file. For procedures that do not appear on the list, Windows leaves their prologue code untouched. However, Windows overwrites the first 3 bytes of all exported procedures with

```
mov     ax, DGROUP
```

where DGROUP represents the selector value for the library's data segment. This explains why the prologue macro reserves the third byte with a **NOP** instruction. The 1-byte instruction serves as padding to provide a 3-byte area for Windows to overwrite.

The epilogue code returns BP to normal, like this:

```
Epilog  MACRO
        pop     ds              ; Recover original DS
        pop     bp              ;   and BP+1
        dec     bp              ; Reset to original BP
        ENDM
```

DLL Data

A DLL can have its own local data segment up to 64K. Besides static data, the segment contains the heap from which a procedure can allocate memory through the **LocalAlloc** API function. You should minimize static data in a DLL to reserve as much memory as possible for temporary allocations. Furthermore, all procedures in the DLL draw from the same heap space. If more than one procedure in the library accesses the heap, a procedure should not hold allocated space unnecessarily at the expense of the other procedures.

A Windows-based program must reserve a "task header" in the first 16 bytes of its data segment. If you link your program with a C run-time function, the C startup code automatically allocates the task header. Otherwise, you must explicitly reserve and initialize the header with zeros. The sample program described in "Example of a DLL:SYSINFO," page 267, shows how to allocate a task header.

DLL Stack

A DLL does not declare a stack segment and does not allocate stack space. A client program calls a library's exported procedure through a simple far call, and the stack does not change. The procedure is, in effect, part of the calling program, and therefore uses the caller's stack.

This simple arrangement differs from that used in small and medium models, in which many C run-time functions accept near pointers as arguments. Such functions assume the pointer is relative to the current data segment. In applications, the call works even if the argument points to a local variable on the stack, since DS and SS contain the same segment address.

However, in a DLL, DS and SS point to different segments. Under small and medium models, a library procedure must always pass pointers to static variables located in the data segment, not to local variables on the stack.

When you write a DLL, include the **FARSTACK** keyword with the **.MODEL** directive, like this:

```
.MODEL  small, pascal, farstack
```

This informs the assembler that SS points to a segment other than DGROUP. With full segment definitions, also add the line:

```
ASSUME  DS:DGROUP, SS:NOTHING
```

DLL Extension Names

You can name an explicitly-loaded DLL file with any extension. The many files in your Windows directory with extensions such as .DRV and .FON are almost certainly DLLs. Many DLLs have an .EXE extension, though they are not true executable files.

A library with an .EXE extension should always include stub code, specified by the **STUB** statement in the module-definition file. The stub code activates when run under MS-DOS, usually displaying a message to inform the user that the program requires Windows. Without the stub code, the system hangs if a user attempts to run a DLL with an .EXE extension.

Do not name a DLL with a .COM extension, since MS-DOS will give control to the first byte of the program header. The header does not contain executable instructions, and the system will hang even if the DLL has stub code.

Summary

Following is a summary of the previous information in this chapter.

- A dynamic-link library has only one instance—that is, it can load only once during a Windows session.
- A single DLL can service calls from many client programs. Windows takes care of linkage between the DLL and each client.
- Windows loads a DLL either implicitly (along with the first client) or explicitly (when the first client calls **LoadLibrary**). It unloads the DLL when the last client either terminates or calls **FreeLibrary**.
- A client calls a DLL routine as a simple far procedure. The routine can use any calling convention.
- Windows ensures that the first instruction in a DLL procedure moves the address of the library's data segment into AX. You must provide the proper prologue code to allow space for this 3-byte instruction and to copy AX to DS.
- All procedures in a DLL have access to a single common data segment. The segment contains both static variables and heap space, and cannot exceed 64K.
- A DLL procedure uses the caller's stack.
- All exported procedures in a DLL must appear in the **EXPORTS** list in the library's module-definition file.

Example of a DLL: SYSINFO

Like any library, a DLL should be as small and fast as possible—a good argument for writing it in assembly language. This section describes an example library called SYSINFO, written entirely in assembly language. The following text applies previous information in this chapter to an actual DLL.

SYSINFO contains three callable procedures. The acronym ASCIIZ refers to a string of ASCII characters terminated with a zero. The callable procedures are:

Procedure	Description
GetSysTime	Returns a far pointer to a 12-byte ASCIIZ string containing the current time in *hh:mm:ss* format.
GetSysDate	Returns a far pointer to an ASCIIZ string containing the current date in any of six languages.
GetSysInfo	Returns a far pointer to a structure containing the following system data: • ASCIIZ string of Windows version • ASCIIZ string of MS-DOS version • Current keyboard status • Current video mode

- Math coprocessor flag
- Processor type
- ASCIIZ string of ROM-BIOS release date

To see SYSINFO in action, follow the steps below. The file SYSDATA.EXE resides in the SAMPLES\WINDLL subdirectory of MASM if you requested example files when installing MASM. Otherwise, you must first install the file with the MASM 6.1 SETUP utility.

- Create SYSINFO.DLL as described in the following section and place it in the SAMPLES\WINDLL subdirectory for MASM 6.1.
- From the Windows File Manager, make the SAMPLES\WINDLL subdirectory the current directory.
- In the Program Manager, choose Run from the File menu and type

  SYSDATA

  to run the example program SYSDATA.EXE. This program calls the routines in SYSINFO.DLL and displays the returned data.

Entry Routine for SYSINFO

SYSINFO links with the DLLENTRY module, which serves as the library's entry point when Windows first loads the program. For a listing and description of DLLENTRY.ASM, see the previous section, "Entry Procedure."

DLLENTRY replaces the LIBENTRY module provided with the Windows SDK, but unlocks the data segment after calling the API function **InitTask**. LIBENTRY does not unlock the segment. DLLENTRY saves some space over LIBENTRY, because it does not pass any arguments to `LibMain`.

The `LibMain` procedure handles the library's initialization tasks. You can name the procedure whatever you want, provided you make the same change in

DLLENTRY.ASM and reassemble both modules. You can even combine DLLENTRY with `LibMain` to form one procedure, like this:

```
DLLInit PROC FAR PASCAL PUBLIC          ; Entry point for DLL

        jcxz    @F                      ; If no heap, skip
        INVOKE  LocalInit, ds, 0, cx    ; Else set up the heap
        .IF     ( ax )                  ; If successful,
        INVOKE  UnlockSegment, -1       ;   unlock the data segment
@@:     .                               ; Initialize DLL data. This
        .                               ;   replaces the call to the
        .                               ;   LibMain procedure.
        mov     ax, TRUE                ; Return AX = 1 if okay,
        .ENDIF                          ;   else if LocalInit error,
        ret                             ;   return AX = 0

DLLInit ENDP
END    DLLInit
```

Whatever you call your combined procedure (`DLLInit` in the preceding example), place the name on the **END** statement as shown. This identifies the procedure as the one that first executes when Windows loads the DLL.

SYSINFO accommodates several international languages. Currently, SYSINFO recognizes English, French, Spanish, German, Italian, and Swedish, but you can easily extend the code to include other languages. `LibMain` calls **GetProfileString** to determine the current language, then initializes the variable `indx` accordingly. The variable indirectly points to an array of strings containing days and months in different languages. The `GetSysDate` procedure uses these strings to create a full date in the correct language.

Static Data

SYSINFO stores the strings in its static data segment. This data remains in memory along with the library's code. All procedures have equal access to the data segment.

Because the library does not call any C run-time functions, it explicitly allocates the low paragraph of the data segment with the variable `TaskHead`. This 16-byte area serves as the required Windows task header, described in "DLL Data," earlier in this chapter.

Module-Definition File

The library's module-definition file, named SYSINFO.DEF, looks like this:

```
LIBRARY         SYSINFO
DESCRIPTION     'Sample assembly-language DLL'
EXETYPE         WINDOWS
CODE            PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE SINGLE
SEGMENTS CODE2  PRELOAD FIXED
EXPORTS         WEP             @1      RESIDENTNAME
                GetSysTime      @2
                GetSysDate      @3
                GetSysInfo      @4
```

Note the following points about the module-definition file:

- The **LIBRARY** statement identifies SYSINFO as a dynamic-link library.
- SYSINFO places its termination procedure `WEP` in a separate code segment, called CODE2, which the **SEGMENTS** statement declares as **FIXED**. This keeps the `WEP` routine fixed in memory, while all other code remains moveable.
- The **EXPORTS** section lists all procedures the library exports, including `WEP`.
- None of the library's procedures require heap space, so SYSINFO.DEF includes no **HEAPSIZE** statement.

Assembling and Linking SYSINFO

The following listing shows the description file for SYSINFO:

```
sysinfo.obj:  sysinfo.asm dll.inc
  ML /c /W3 sysinfo.asm
dllentry.obj:  dllentry.asm dll.inc.
  ML /c /W3 dllentry.asm
sysinfo.dll: dllentry.obj sysinfo.obj
  LINK dllentry sysinfo, sysinfo.dll,, libw.lib mnocrtdw.lib,
sysinfo.def
```

To create SYSINFO.DLL, run the **NMAKE** utility described in Chapter 16 of *Environments and Tools*. Or assemble and link SYSINFO with the three command lines shown in the preceding listing. This does not require running **NMAKE**.

SYSINFO links with the library modules MNOCRTDW.LIB and LIBW.LIB. The first supplies the required Windows startup code for a medium-model DLL that does not use any C run-time functions. LIBW.LIB is the Windows import library, which contains no executable code. The import library provides linkage information for the Windows API functions referenced in the DLL. Windows establishes the final links when it loads the program.

Expanding SYSINFO

SYSINFO is an example of how to write an assembly-language DLL without overwhelming detail. It has plenty of room for expansion and improvements. The following list may give you some ideas:

- To create a heap area for the library, add the line

  HEAPSIZE *value*

  to the module-definition file, where value is an approximate guess for the amount of heap required in bytes. The `DLLEntry` procedure automatically allocates the indicated amount of heap. Keep the data segment moveable, because Windows then provides more heap space if required by the DLL procedures.

- If you want to add a procedure that calls C run-time functions, you must replace MNOCRTDW.LIB with MDLLCW.LIB, which is supplied with the Windows SDK. The MDLLCW.LIB library contains the run-time functions for medium-model DLLs.

- Each time the `GetSysInfo` procedure is called, it retrieves the version number of MS-DOS and Windows, gets the processor type, checks for a coprocessor, and reads the ROM-BIOS release date. Since this information does not change throughout a Windows session, it would be handled more efficiently in the LibMain procedure, which executes only once. The code is currently placed in `GetSysInfo` for the sake of clarity at the expense of efficiency.

- SYSINFO is not a true international program. You can easily add more languages, extending the `days` and `months` arrays accordingly. Moreover, for the sake of simplicity, the `GetSysDate` procedure arranges the date with an American bias. For example, in many parts of the world, the date numeral appears before the month rather than after. If you use SYSINFO in your own applications, you should include code in `LibMain` to determine the correct date format with additional calls to **GetProfileString**. You can find more information on how to do this in Chapter 18 of the Microsoft Windows *Programmer's Reference, Volume 1,* supplied with the Windows SDK.