Microsoft® CodeView® and Utilities

Software Development Tools

for MS® OS/2 and MS-DOS® Operating Systems

Update

Microsoft Corporation

Document No. 614010011-000-R00-0288
Part No. 01701
10 9 8 7 6 5 4 3 2 1

Contents

Figures

Section 1
Introduction

This update supplements the Microsoft® CodeView® and Utilities manual, and describes utilities that are designed for use with Microsoft Windows and the OS/2 systems. The update also describes improvements which apply to both real-mode and protected-mode environments, including new features of the Microsoft CodeView debugger. The pages that follow use the term "OS/2" to refer to both Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" is used to refer to both MS-DOS® and IBM Personal Computer DOS.

The development of a protected-mode program under OS/2 differs from the development of a real-mode program in the following way: to make calls to OS/2 you must call a dynamic-link library. (As explained in Section 3, "About Linking in OS/2," a dynamic-link library is not linked to the program but is loaded separately at run time.) The use of a dynamic-link library, in turn, requires that the program know where its dynamic-link functions are defined. Module-definition files and import libraries, described below, serve this purpose. OS/2 programs can also take advantage of multiple threads, which are parts of your program that run concurrently. (Threads are like processes, but they are faster to create, and they share the same code segment.)

The following list describes what you can do with the new utilities and the new version of the CodeView debugger:

- **View structures with the CodeView debugger**

  In addition to the capabilities described in the Microsoft CodeView and Utilities manual, both versions of the debugger (protected mode and real mode) provide the ability to watch a C or MASM structure, Pascal record, or BASIC user-defined type. The debugger displays, labels, and dynamically updates each element of the structure, and allows you to trace through a linked list with a simple keystroke or mouse selection.

- **Debug multiple-thread programs with CVP**

  The protected-mode CodeView debugger, CVP, expands the capabilities of the debugger as described in the Microsoft CodeView and Utilities manual. The protected-mode debugger can debug code in dynamic-link libraries, and it helps you debug multiple-thread programs by providing a new command. This command lets you view the state of the machine while one thread or another is being traced. You can also freeze some threads while the others run concurrently.

- **Link real- and protected-mode programs**

  Version 5.0 of the Microsoft Segmented-Executable Linker (**LINK**) takes an additional field for module-definition files (module-definition files are documented in Section 7). The use of a module-definition file makes it possible for you to create dynamic-link libraries, specify dynamic-link entry points for functions, and provide other kinds of information for your OS/2 program modules.

- **Create import libraries with IMPLIB**

  Import libraries (described in Section 3, "About Linking in OS/2," and Section 6, "The IMPLIB Utility") can speed up the development process for OS/2 applications. When you create a dynamic-link library, you can provide an import library to the application developer who calls your dynamic-link library. The import library is easy to link, and saves the developer the trouble of creating a module-definition file. The Microsoft Import Library Manager utility (**IMPLIB**) generates import libraries for you.

- **Use BIND to create dual-mode applications**

  By using the Microsoft Operating System/2 Bind utility (**BIND**), you can convert an OS/2 program so that it can run in either OS/2 protected mode or in real mode (DOS 3.x or compatibility box). Section 5 gives instructions for using the **BIND** utility.

- **Link faster with ILINK**

  For large OS/2 and Windows programs, the Microsoft Incremental Linker (**ILINK**) can speed up linking by as much as 20 times. This utility takes advantage of the new segmented-executable file format, by relinking only those modules which have changed. Section 9 explains how the process works.

1.1 System Requirements

To use all of the utilities presented in this manual, you need to have MS OS/2 installed and running in protected mode.

In addition, if you want to call the operating system or take advantage of OS/2-specific features such as threads or semaphores, you will need to have documentation on all the Application Program Interface (API) calls (see the *Microsoft Operating System/2 Programmer's Reference*).

The following programs will also run in real mode (DOS 3.x or OS/2 compatibility box):

- **LINK**
- **CV** (but not **CVP**)
- **MAKE**
- **ILINK**

1.2 Installation

The MS OS/2 languages include two versions of the Microsoft CodeView debugger, one for each OS/2 operating mode. For debugging programs running in the protected mode, the CodeView debugger's executable file is **CVP.EXE**, and the help file is **CVP.HLP**. Both should be installed in a directory listed in the **PATH** environment variable. To debug programs running in real mode, use the **CV.EXE** executable file and its help file, **CV.HLP**. Both of these files should also be installed in a directory listed in the **PATH** environment variable.

Finally, you should also install the executable files **LINK**, **EXEC**, **ILINK**, **BIND**, and **IMPLIB** in a directory listed in the **PATH** environment variable.

Note

This document uses certain notational conventions to convey example and syntax information for various utilities. See the "Introduction" to the Microsoft CodeView and Utilities manual for an explanation of these conventions.

Within this update, command-line options are preceded by a forward slash (/). However, in all cases where a slash is used, you can enter either a slash or a dash (-).

Section 2
Using the CodeView Debugger

This chapter first presents two new features—structure watching and text selection—which are included in both the protected-mode CodeView debugger (**CVP.EXE**) and the real-mode CodeView debugger (**CV.EXE**). The chapter then describes the special features that are included only in the protected-mode debugger. Finally, the chapter describes the Microsoft Debug Information Compactor utility (**CVPACK**), which reduces the size of the executable file.

2.1 New Debugging Features

The CodeView debugger now provides two direct ways to examine values of members of structures. First, you can now specify a structure name in a Watch command or Evaluate Expression command (see Section 2.1.1, "Placing Structures in the Watch Window"). Second, the debugger provides a new command for viewing structures in a dialog box. This new command is described in Section 2.1.2, "Using the Graphic Display Command."

Note

For ease of discussion, Section 2.1.1, "Placing Structures in the Watch Window," uses the general term "structures" to refer to Pascal records and BASIC user-defined types, as well as C structures. The machine-level implementation of these records, types, and structures is similar, so the debugger handles them in similar ways.

The debugger also provides text selection, which permits you to use a mouse (Microsoft Mouse or compatible) to select text on screen as input to commands. This capability is described in Section 2.1.3, "Selecting Text."

2.1.1 Placing Structures in the Watch Window

Assume that you have declared a structure as follows:

```
struct stype {
int   a;
int   b;
struct {
    int   x;
    long  y;
} c;
    struct stype *new;
} sample = {11, 12, {100, 200} };
```

If you give the Watch command `W?sample`, then the debugger displays the following line in the watch window:

```
sample : { a=11, b=12, c={x=100, y=200}, new=0x0000:0x0000 }
```

Note the following features, as shown in the above example:

- Nested structures are displayed in a nested pair of braces. (The debugger displays structures nested to any level!)
- Fields other than nested structures are displayed in their default format, as described in the Microsoft CodeView and Utilities manual. For example, a pointer is always displayed in the standard *segment:offset* form. (The example above assumes the C hexadecimal notation.)

2.1.2 Using the Graphic Display Command

The new Graphic Display command (**??**) is even more powerful than the Watch and Evaluate Expression commands. This command is especially useful for examining nested structures and linked lists of pointers. The syntax of the command is simple:

?? *variable*[[**,c**]]

In the syntax display above, *variable* can be any recognized data symbol. The optional format specifier **c** can be used to specify that one-byte-length fields be displayed as ASCII (American Standard Code for Information Interchange) characters.

The debugger responds by displaying a dialog box. If *variable* is a structure, then the dialog box contains the name and value of each field. For example, if the structure `sample` is defined as described in the previous section, then the command `??sample` produces the following dialog box:

```
                                x
-------------------------------------------------------
a                               11
b                               12
c                               {...}
new                             0x0000:0x0000
```

Nested structures, such as c in the example above, are evaluated as {...}. In addition, the dialog box displays a null-terminated ASCII string next to any field which contains a character pointer.

You can use the Graphic Display command with variables other than structures. With nonstructure variables, the command displays just one field.

The Graphic Display command enables you to expand structures and dereference pointers by selecting a field. (These actions are defined below.) To select a field with the keyboard, press the up and down DIRECTION keys to move the cursor to the field you wish to select, and then press ENTER. To select a field with the mouse, simply click the left mouse button on the field you wish to select (or anywhere on the same line). Selecting a field has the following effect:

1. If the field contains a nested structure, then the structure is "expanded"; the nested structure becomes the new subject of the dialog box. The dialog box displays each field of the nested structure.

2. If the field contains a pointer, then the pointer is "dereferenced"; in other words, the debugger locates the data which the pointer addresses. This data becomes the new subject of the dialog box.

   The pointer's type determines how the debugger displays the dereferenced data. The debugger uses this type information even if the pointer does not currently address any meaningful data. If the pointer addresses a structure, then each field of the new structure is displayed.

3. If the field contains neither a pointer nor a nested structure, then selection has no effect. The debugger beeps to tell you that the selected field was neither a pointer nor a structure.

You can return to the previous dialog-box display (before expansion or dereference took place) by pressing the backspace key or by clicking right (press the righthand mouse button).

While the dialog box is on screen, you cannot execute other CodeView-debugger commands. To remove the dialog box and resume normal debugger operation, press ESC, or click left while the mouse cursor is outside the box.

Note

You can take advantage of the new Watch-command capability in either window or sequential mode. To use the Graphic Display command, however, you need to run the debugger in window mode.

2.1.3 Selecting Text

Text selection is a technique that you can use with the mouse. Select text from either the display or dialog window by holding down the left mouse button and dragging the mouse cursor to the left or right. All text up to the mouse cursor is selected when you release the button. Once selected, you can use the text in one of two ways:

1. The selected text automatically appears in the next dialog prompt box. For example, when you select Find from the Search menu, a dialog box prompts you for the search string. Your selected text appears in this box. You can edit the text or press ENTER immediately.

2. The selected text appears in the dialog window (at the end of the dialog-window buffer) when you press SHIFT+INS. If you then press ENTER, the text is given to the debugger as a command.

The selected text can only be used once. To use the same text repeatedly, you need to reselect the text after each use.

2.2 The Protected-Mode CodeView Debugger

The protected-mode CodeView debugger (**CVP.EXE**) differs from the real-mode CodeView debugger (**CV.EXE**) in three principal ways:

1. The View Output Screen command (\\) works differently.

2. **CVP** takes an additional command-line option for use in debugging dynamic-link modules.

3. **CVP** can debug multiple-thread programs. In order to deal with the multiple-thread capability of OS/2, **CVP** has a new command that is not present in **CV**, and some of the commands for tracing and execution in **CV** work differently in **CVP**.

Each of these differences is described in the sections below. You should also bear in mind the following general limitations when using **CVP** in the OS/2 environment:

- Only one copy of the CodeView debugger can be run at a time in the protected mode. Multiple copies cannot be run in concurrent screen groups.
- When you debug a program without using the /2 option, and the program makes dynamic-link calls to functions outside the **API**, the debugger will not have access to the program's environment or the current drive and directory.

In all other respects, the CodeView debugger's operation as described in the Microsoft CodeView and Utilities manual applies to both versions.

2.2.1 Using the Debugger's View Output Command

When you switch display to the output window with **CVP**, by using the View Output command (\), you won't stay there indefinitely as you would with the real-mode CodeView debugger. Instead, you will jump back to the CodeView screen after a 3-second delay. A different delay period (as measured in seconds) can be specified with a number following the View Output command, as in the following example:

```
\60
```

The example above directs the debugger to display the output window for 60 seconds, before returning to the debugging screen.

Another way to view the output is to go back to the Session Manager screen and select the screen group labeled **CVP APP**. This is the screen group owned by the application that is being debugged. When you have finished viewing the output window, switch back to the **CVP.EXE** screen group. You can use ALT+ESC to toggle between screen groups.

2.2.2 Debugging Dynamic-Link Modules

The protected-mode CodeView debugger (**CVP**) can debug dynamic-link modules, but only if it is told what libraries to search at run time. For more information on dynamic-link libraries, refer to the *Microsoft Operating System/2 Programmer's Guide*, and to the **IMPLIB** and module-definition sections in this update (Sections 6 and 7, respectively).

When you place a module in a dynamic-link library, neither code nor symbolic information for that module is stored in the executable (.EXE) file; instead, the code and symbols are stored in the library and are not brought together with the main program until run time.

Thus, the protected-mode debugger needs to search the dynamic-link library for symbolic information. Because the debugger does not automatically know what libraries to look for, **CVP** has an additional command-line option which enables you to specify dynamic-link libraries:

■ Syntax

/L *file*

The **/L** option directs the CodeView debugger to search *file* for symbolic information. When you use this option, at least one space must separate **/L** from *file*.

■ Example

```
CVP /L DLIB1.DLL /L GRAFLIB.DLL PROG
```

In the example above, **CVP** is invoked to debug the program **PROG.EXE**. To find symbolic information needed for debugging each module, **CVP** will search the libraries `DLIB1.DLL` and `GRAFLIB.DLL`, as well as the executable file `PROG.EXE`.

2.2.3 Debugging Multiple-Thread Programs

A program running in OS/2 protected mode has one or more threads. As explained in the programmer's guide, threads are the fundamental units of execution; OS/2 can execute a number of different threads concurrently. A thread is similar to a process, yet it can be created or terminated much faster. Threads begin at a function-definition heading, in the same program in which they are invoked.

The existence of multiple threads within a program presents a dilemma for debugging. For example, thread 1 may be executing source line 23 while thread 2 is executing source line 78. Which line of code does the CodeView debugger consider to be the current line?

Conversely, you cannot always tell which thread is executing just because you know what the current source line is. In OS/2 protected mode, you can write a program in which two threads enter the same function.

In Figure 2.1, the function `main` uses the **DOSCREATETHREAD** system call to begin execution of thread 2. The function `f2` is the entry point of the new thread. Thread 2 begins and terminates inside the function `f2`. Before it terminates, however, thread 2 can enter other functions by means of ordinary function calls.

Thread 1 begins execution in the function `main`, and thread 2 begins execution in the function `f2`. Later, both thread 1 and thread 2 enter the function `f3`. (Note that each thread returns to the proper place because each thread has its own stack.) When you use the debugger to examine the behavior of code within the function `f3`, how can you tell which thread you are tracking?

The protected-mode CodeView debugger solves this dilemma by using a modified CodeView prompt, and by providing the Thread command, which is only available with **CVP**.

The command prompt for the protected-mode CodeView debugger is preceded by a three-digit number indicating the current thread.

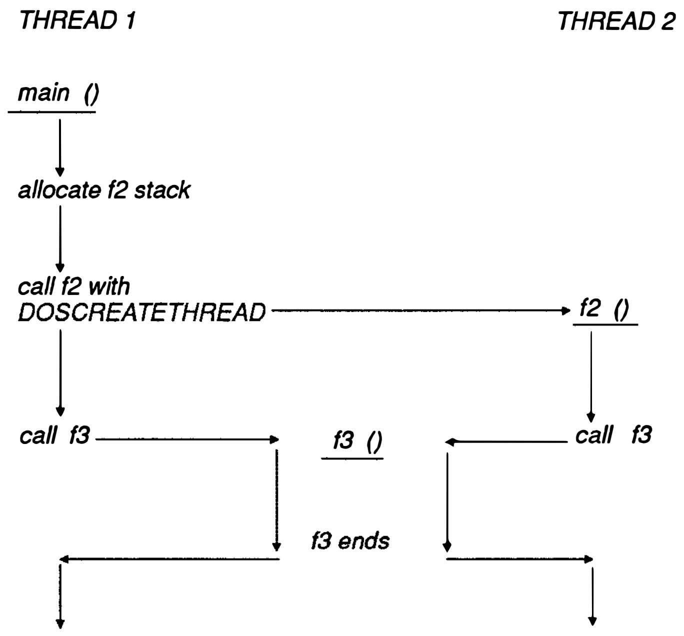

Figure 2.1 Multiple-Thread Program

■ **Example**

```
001>
```

The example above displays the protected-mode CodeView prompt, indicating that thread 1 is the current thread. Thread 1 is always the current thread when you begin a program. If the program never calls the **DOSCREATETHREAD** function, then thread 1 will remain the only thread.

Each thread has its own stack and its own register values. When you change the current thread, you will see several changes to the CodeView-debugger display:

- The CodeView prompt will display a different three-digit number.

- The register contents will all change.
- The current source line and current instruction will both change, to reflect the new value of **CS:IP**. If you are running the debugger in window mode, you will likely see different code in the display window.
- The Calls menu and the Stack Trace command will display a different group of functions.

The rest of this section discusses the Thread command, and lists other CodeView commands that may work differently because of multiple threads.

■ Syntax

The syntax of the Thread command is displayed below:

~⟦*specifier*⟦*command*⟧⟧

In the syntax display above, the *specifier* determines to which thread or threads the command will apply. You can specify all threads, or just a particular thread. The *command* determines what activity the debugger will carry out with regard to the specified thread. For example, you can execute the thread, freeze its execution, or select it as the current thread. If you omit *command*, the debugger displays the status of the specified thread. If you omit both the *command* and *specifier*, then the debugger displays the status of all threads.

The status display for threads consists of the two fields

thread-id   *thread-state*

in which *thread-id* is an integer, and *thread-state* has the value `runnable` or `frozen`. All threads not frozen by the debugger are displayed as `runnable`; this includes threads that may be blocked for reasons that have nothing to do with the debugger, such as a critical section.

The legal values for *specifier* are listed below, along with their effects.

Symbol	Function
(blank)	Displays the status of all threads. If you omit the *specifier* field you cannot enter a *command.* Instead, you simply enter the tilde (~) by itself.
#	Specifies the last thread that was executed. This thread is not necessarily the current thread. For example, suppose you are tracing execution of thread 1, and

	then switch the current thread to thread 2. Until you execute some code in thread 2, the debugger still considers thread 1 to be the last thread executed.
*	Specifies all threads.
n	Specifies the indicated thread. The value of *n* must be a number corresponding to an existing thread. You can determine corresponding numbers for all threads by entering the command `~*`, which gives status of all threads.
.	Specifies the current thread.

The legal values for *command* are listed below, along with their meanings.

Command	Function
(blank)	The status of the selected thread (or threads) is displayed.
BP	A breakpoint is set for the specified thread or threads. As explained earlier, it is possible to write your program so that the same function is executed by more than one thread. By using this version of the Thread command, you can specify a breakpoint that applies only to a particular thread. The letters **BP** are followed by the normal syntax for the Breakpoint Set command, as described in the Microsoft CodeView and Utilities manual. Therefore you can include the optional passcount and command fields.
E	The specified thread is executed in slow motion. When you specify a single thread with **E**, then the specified thread becomes the current thread, and is executed without any other threads running in the background. The command `~*E` is a special case. It is legal only in source mode, and executes the current thread in slow motion, but lets all other threads run (except those that are frozen). You will only see the current thread executing in the debugger display.
F	The specified thread (or threads) is frozen. A frozen thread will not run in the background or in response to the debugger Go command. However, if you use the **E**, **G**, **P**, or **T** variation of the Thread command, then the specified thread will be temporarily unfrozen while the debugger executes the command.

G	Control is passed to the specified thread, until it terminates or until a breakpoint is reached. If you give the command `~*G`, then all threads will execute concurrently (except for those that are frozen). If you specify a particular thread, then the debugger will temporarily freeze all other threads and execute the specified thread.
P	The debugger executes a program step for the specified thread. If you specify a particular thread, then the debugger executes one source line or instruction of the thread. All other threads are temporarily frozen. This version of the Thread command does not change the current thread. Therefore if you specify a thread other than the current thread, you will not see immediate results. However, the subsequent behavior of the current thread may be affected. The command `~*P` is a special case. It is legal only in source mode, and causes the debugger to step to the next source line, while letting all other threads run (except for those that are frozen). You will only see the current thread execute in the debugger display.
S	The specified thread is selected as the current thread. This version of the Thread command can apply to only one thread at a time. Thus, the command `~*S` results in an error message. Note that the command `~.S` is legal, but has no effect.
T	The specified thread is traced. This version of the Thread command works in a manner identical to **P**, described above, except that **T** traces through function calls and interrupts, whereas **P** does not.
U	The specified thread or threads are unfrozen. This command reverses the effect of a freeze.

Note

With the Thread command, only the **S** (select) and the **E** (execute) variations cause the debugger to switch the current thread. However, when a thread causes program execution to stop by hitting a breakpoint, the debugger will select that thread as the current thread.

You can prevent the debugger from changing the current thread, by including the breakpoint command `"~.S"`. This command directs the debugger to switch to the current thread rather than the thread that hit the breakpoint. For example, the following command sets a breakpoint at line `120` and prevents the current thread from changing:

```
BP .120 "~.S"
```

■ Syntax

The syntax display below summarizes all the possible entries to the Thread command:

~{#|*|*n*|.}[[BP|E|F|G|P|S|T|U]]

Note that you must include one of the symbols from the first set (which gives possible values for the specifier), but you do not have to include a symbol from the second set (which gives possible values for the command).

■ Examples

```
004>~
```

The example above displays the status of all threads, including their corresponding numbers.

```
004>~2
```

The example above displays the status of thread 2.

```
004>~5S
```

The example above selects thread 5 as the current thread. Since the current thread was 4 (a fact apparent from the CodeView prompt), this means that the current thread is changing and therefore we can expect the registers and the code displayed to all change.

```
005>~3BP .64
```

The example above sets a breakpoint at source line 64, which stops program execution only when thread 3 executes to this line.

```
005>~1F
```

The example above freezes thread 1.

```
005>~*U
```

The example above thaws (unfreezes) all threads; any threads that were frozen before will now be free to execute whenever the Go command is given. If no threads are frozen, this command has no effect.

```
005>~2E
```

The example above selects thread 2 as the current thread, then proceeds to execute thread 2 in slow motion.

```
002>~3S
003>~.F
003>~#S
002>
```

The example above selects thread 3 as the current thread, freezes the current thread (thread 3), and then switches back to thread 2. After we switched to thread 3, no code was executed; therefore the debugger considers the last-thread-executed symbol (#) to refer to thread 2.

Whether or not you use the Thread Command, the existence of threads affects your CodeView debugging session at all times. Particular debugger commands are strongly affected. Each of these commands is discussed below.

Command	**Behavior in Multiple-Thread Programs**
.	The Current Line command always uses the current value of **CS:IP** to determine what the current instruction is. Thus, the Current Line command applies to the current thread.
E	When the debugger is in source mode, the Execute command is equivalent to the `~*E` command; the current thread is executed in slow motion while all other threads are also running. When the debugger is in mixed or assembly mode, the Execute command is equivalent to the command `~.P`, which does not let other threads run concurrently.
BP	The Set Breakpoint command is equivalent to the `~*BP` command; the breakpoint applies to all threads.

G	The Go command is equivalent to the `~*G` command; control is passed to the operating system, which executes all threads in the program except for those that are frozen.
P	When the debugger is in source mode, the Program Step command is equivalent to the command `~*P`, which lets other threads run concurrently. When the debugger is in mixed or assembly mode, the Program Step command is equivalent to the command `~.P`, which lets no other threads run.
K	The Stack Trace command displays the stack of the current thread.
T	When the debugger is in source mode, the Trace command is equivalent to the command `~*T`, which lets other threads run concurrently. When the debugger is in mixed or assembly mode, the Trace command is equivalent to the command `~.T`, which lets no other threads run.

In general, CodeView-debugger commands apply to all threads, unless the nature of the command makes it appropriate to deal with only one thread at a time. (For example, since each thread has its own stack, the Stack Trace command does not apply to all threads.) In the later case, the command applies to the current thread only.

2.3 Saving Memory with the CVPACK Utility

After you compile and link a program with CodeView debugging information, you can use the Microsoft Debug Information Compactor utility (**CVPACK**) to reduce the size of the executable file. **CVPACK** compresses the debugging information in the file, and allows the CodeView debugger to load larger programs without running out of memory.

The **CVPACK** utility has the following command line:

CVPACK [[**/p**]] *exefile*

The **/p** option results in the most effective possible packing, but causes **CVPACK** to take longer to execute. When the **/p** option is specified, unused debugging information is discarded, and the packed information is sorted within the file. When the **/p** option is not specified, packed information is simply appended to the end of the file.

To debug a file that has been altered with **CVPACK**, you must use Version 2.10 or later of the CodeView debugger.

Section 3
About Linking in OS/2

In most respects, linking a program using the Microsoft Segmented-Executable Linker Version 5.0 (**LINK**) for the OS/2 environment is similar to linking a program for the DOS 3.x environment. The principal difference is that most programs created for the DOS 3.x environment run as stand-alone applications, whereas programs that run under OS/2 protected mode generally call one or more “dynamic-link libraries.”

A dynamic-link library contains executable code for common functions, just as an ordinary library does. Yet code for dynamic-link functions is not linked into the executable (**.EXE**) file. Instead, the library itself is loaded into memory at run time, along with the **.EXE** file.

Each **.DLL** file (dynamic-link library) must use “export definitions” to make its functions directly available to other modules. At run time, functions not exported can only be called from within the same file. Each export definition specifies a function name.

Conversely, the **.EXE** file must use “import definitions” that tell where each dynamic-link function can be found. Otherwise, OS/2 would not know what dynamic-link libraries to load when the program is run. Each import definition specifies a function name and the **.DLL** file where the function resides.

Assume the simplest case, in which you create one application and one dynamic-link library. The linker requires export and import definitions for all dynamic-link function calls. The OS/2 operating system provides two ways to supply these definitions:

1. You create one module-definition file (**.DEF** extension) with export definitions for the **.DLL** file, and another module-definition file with import definitions for the **.EXE** file. The module-definition files provide these definitions in an ASCII format.

2. You create one module-definition file (**.DEF** extension) for the **.DLL** file, and then generate an import library to be linked to the **.EXE** file.

The next two sections consider each of these methods in turn.

3.1 Linking without an Import Library

Figure 3.1 illustrates the first way to supply definitions for dynamic-link function calls, in which each of the two files—the **.DLL** file and the **.EXE** file—has a corresponding module-definition file. (A module-definition file has a **.DEF** default extension.)

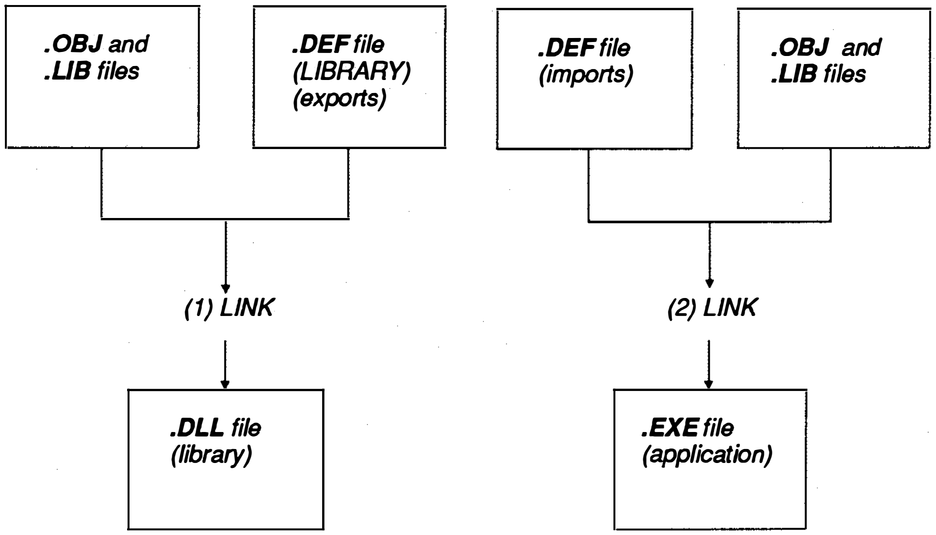

Figure 3.1 Linking without an Import Library

The two major steps may be described as follows:

1. Object files (and possibly standard-library files) are linked together with a module-definition file to create a dynamic-link library. A module-definition file for a dynamic-link library has at least two statements. The first is a **LIBRARY** statement, which directs the linker to create a **.DLL** rather than an **.EXE** file. The second statement is a list of export definitions.

2. Object files (and possibly standard-library files) are linked together with a module-definition file to create an application. The module-definition file for this application contains a list of import definitions. Each definition in this list contains both a function name and the name of a dynamic-link library.

The DOS 3.x linker has no way to accept a module-definition file as input. However, the dual-mode (OS/2) linker has an additional field for a module-definition file. This field is discussed in Section 4, "Using the OS/2 Linker."

3.2 Linking with an Import Library

Figure 3.2 illustrates the second way to supply definitions for dynamic-link function calls, in which a module-definition file is supplied for the dynamic-link library, and an import library is supplied for the application.

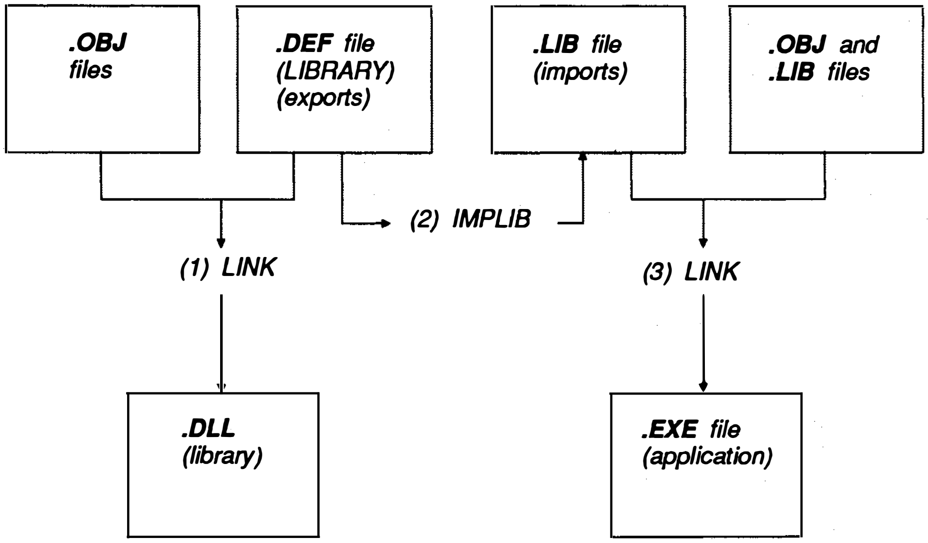

Figure 3.2 Linking with an Import Library

The three major steps may be explained as follows:

1. Object files are linked to produce a **.DLL** file. This step is identical to the first step in the previous section. Note that the module-definition file contains export definitions.

2. The **IMPLIB** utility is used to generate an import library. **IMPLIB** takes as input the same module-definition file used in the first step. **IMPLIB** knows the name of the library module (which by default has the same base name as the **.DEF** file), and it determines the name of each exported function by examining

export definitions. For each export definition in the **.DEF** file, **IMPLIB** generates a corresponding import definition.

3. The **.LIB** file generated by **IMPLIB** is used as input to **LINK**, which creates an application. This **.LIB** file does not use the same file format as a **.DEF** file, but it fulfills the same purpose: to provide the linker with information about imported dynamic-link functions.

The **.LIB** file generated by **IMPLIB** is called an import library. Import libraries are similar in most respects to ordinary libraries; you specify import libraries and ordinary libraries in the same command-line field of **LINK**, and you can append the two kinds of libraries together (by using the Library Manager). Furthermore, both kinds of libraries resolve external references at link time. The only difference is that import libraries do not contain executable code, merely records that describe where the executable code can be found at run time.

So far, only simple scenarios have been considered. Dynamic linking is flexible, and supports much more complicated scenarios. An application can make calls to more than one dynamic-link library. Furthermore, module-definition files for libraries can import functions as well as export them. It is perfectly possible for a **.DLL** file to call another **.DLL** file, and so on, to any level of complexity; the result may be a situation in which many files are loaded at run time.

3.3 Why Use Import Libraries?

At first glance, it may seem easier to create programs without import libraries, since import libraries add an extra step to the linking process. Usually, however, it is easier to use import libraries. There are two reasons why this is so.

First, the **IMPLIB** utility automates much of the program-creation process for you. To run **IMPLIB**, you specify the **.DEF** file that you already created for the dynamic-link library. Operation of **IMPLIB** is simple. If you do not use an import library generated by **IMPLIB**, then you must use an ASCII text editor to create a second **.DEF** file, where you explicitly give all needed import definitions.

Second, the first two steps in the linking process described above (creation of the **.DLL** file and creation of the import library) may be carried out only by the author of the dynamic-link library. The libraries may then be given to an applications programmer, who focuses on linking the application (third step). The application programmer's task is simplified if he links with the import library, because then he does not have to worry about editing his own **.DEF** file. The import library comes ready to link.

A good example of a useful import library is the file **DOSCALLS.LIB**. Protected-mode applications generally need to call one of the dynamic-link system libraries that are released with OS/2; the **DOSCALLS.LIB** file contains import definitions for all calls to these system libraries. It is much easier to link with **DOSCALLS.LIB** than to create a **.DEF** file for every OS/2 program you link.

3.4 Advantages of Dynamic Linking

Why use dynamic-link libraries at all? Dynamic-link libraries serve much the same purpose that standard libraries do, but in addition, dynamic-link libraries give you the following advantages:

1. Link applications faster.

   With dynamic linking, the executable code for a dynamic-link function is not copied into the application's **.EXE** file. Instead, only an import definition is copied. Therefore, linking is usually a bit faster.

2. Save significant disk space.

   Suppose you create a library function called `printit`, and that this function is called by many different programs. If `printit` is in a standard library, then the function's executable code must be linked into each **.EXE** file that calls the function. In other words, the same code resides on your disk in many different files. But if `printit` is stored in a dynamic-link library, then the executable code resides in just one file—the library itself.

3. Make libraries and applications more independent.

   Dynamic-link libraries can be updated any number of times, without relinking the applications that use them. If you are a user of third-party libraries, this fact is particularly convenient. You receive the updated **.DLL** file from the third-party developers, and you need only copy the new library onto your disk. At run time, your applications will automatically call the updated library functions.

4. Utilize shared code and data segments.

   Code and data segments loaded in from a dynamic-link library can be shared. Without dynamic linking, this sharing is not possible because each file has its own copy of all the code and data it uses. By sharing segments with dynamic linking, you can utilize memory much more efficiently.

Section 4
Using the OS/2 Linker

This section describes how to link applications and dynamic-link libraries, and assumes that you are familiar with the concepts of dynamic linking, import libraries, and module-definition files. If you are not familiar with these concepts, then read the previous section, "About Linking in OS/2."

The linker can produce either an application that runs under DOS 3.x, an application that runs under OS/2 (or Microsoft Windows), or a dynamic-link library. The following rules determine what output the linker produces:

1. If no module-definition file or import library resolves any external references, then the linker produces an application for DOS 3.x (In other words, the linker creates a DOS 3.x application unless you specify a module-definition file or import library, and that file resolves at least one external reference.)

2. If a module-definition file with a **LIBRARY** statement is given, then the linker produces a dynamic-link library for OS/2.

3. Otherwise, the linker produces an application for OS/2.

You can therefore produce an OS/2 application by linking with an import library or a module-definition file, as long as you do not use a **LIBRARY** statement. (The **LIBRARY** statement is described in Section 7, "Using Module-Definition Files.") The file **DOSCALLS.LIB** is an import library. Thus, if you link with **DOSCALLS.LIB**, you produce either an OS/2 application or a dynamic-link library.

Note

Throughout this chapter, all references to OS/2 protected mode also apply to Microsoft Windows.

The linker produces files that run in protected mode only or in real mode only. However, OS/2 applications that make dynamic-link calls only to the Family API (a subset of the functions defined in **DOSCALLS.LIB**) can be made to run under DOS 3.x with the **BIND** utility. The **BIND** utility is discussed in the next section.

■ Syntax

Use the following command-line syntax to invoke the OS/2 linker:

LINK *objects* ⟦, ⟦*exe*⟧ ⟦, ⟦*map*⟧ ⟦, ⟦*lib*⟧ ⟦, *def*⟧⟧⟧⟧⟦;⟧

Each of the command-line fields is explained below. In the list that follows, reference is made to libraries. Unless qualified by the term "dynamic-link," the word "libraries" refers to import libraries and standard (object-code) libraries, both of which have the default extension .LIB. (Note that dynamic-link libraries have the default extension .DLL, and therefore are usually easy to tell from other libraries.) You can specify import libraries anywhere you can specify standard libraries. You can also combine import libraries and standard libraries by using the Library Manager; these combined libraries can then be specified in place of standard libraries.

Field	Description
objects	The name of one or more object-code files, to be linked into the application or dynamic-link library. Object files are output by compilers and assemblers. To specify more than one object file, separate each file name by a space or by the plus sign (+). Libraries can also be specified in this field, in which case they are considered "load libraries" by the linker. All objects in a load library (functions and data) are automatically linked into the linker's output.
exe	The name you wish the application or dynamic-link library to have. The default for an application name is the base name of the first object module on the command line, combined with an **.EXE** extension. The default for a dynamic-link-library name is the base name of the module-definition file, combined with a **.DLL** extension. Different defaults may be specified in the module-definition file.
map	The name you wish the map file to have.
libraries	The name of one or more library files, which **LINK** searches to resolve external references. You can also enter directories in this field; **LINK** searches the specified directories in order to find any libraries that it cannot find in the current directory. If you have more than one entry in this field, separate each entry by a space.

def	File name of a module-definition file. The use of a module-definition file is optional for applications, but required for dynamic-link libraries.

Note

The OS/2 linker supports overlays only when producing a real-mode application.

As with the DOS 3.x linker, you may specify command-line options after any field—but before the comma that terminates the field. The rest of this section discusses linker command-line options.

4.1 Options for Real Mode Only

Most of the options listed in Chapter 12 of the Microsoft CodeView and Utilities manual can be used with either protected-mode or real-mode programs. However, the following options can be used only when linking real-mode programs:

Option	Minimum Abbreviation
/CPARMAXALLOC	/CP
/DSALLOCATE	/DS
/HIGH	/HI
/NOGROUPASSOCIATION	/NOG
/OVERLAYINTERRUPT	/O

4.2 Options for Protected Mode Only

The OS/2 linker supports two new options that can be used only when linking protected-mode programs (or with Microsoft Windows applications). As mentioned above, most options described in Chapter 12 of the Microsoft CodeView and Utilities manual can be used for both protected-mode and real-mode programs.

■ **Syntax**

/A[[**LIGNMENT**]]**:***size*

The **/ALIGNMENT** option directs **LINK** to align segment data in the executable file along the boundaries specified by *size*. The *size* argument must be a power of two. For example,

```
ALIGNMENT:16
```

indicates an alignment boundary of 16 bytes. The default alignment for OS/2-application and dynamic-link segments is 512. The minimum abbreviation for this option is **/A**.

■ **Syntax**

/W[[**ARNFIXUP**]]

The **/WARNFIXUP** option directs the linker to issue a warning for each segment-relative fixup of location-type "offset," such that the segment is contained within a group but is not at the beginning of the group. The linker will include the displacement of the segment from the group in determining the final value of the fixup, contrary to what happens with DOS executable files. The minimum abbreviation for this option is **/W**.

4.3 New Options for Both Modes

In addition to the options listed in Chapter 12 of the Microsoft CodeView and Utilities manual, the OS/2 linker also supports the following options for both real-mode and protected-mode programs. The **/NONULLSDOSSEG** option is primarily of interest to Windows programmers, as is the **/W** option described above.

■ **Syntax**

/NOE[[**XTENDEDDICTSEARCH**]]

The **/NOEXTENDEDDICTSEARCH** option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains. Normally, the linker consults this list to speed up library searches. The effect of the **/NOE** option is to slow down the linker. You often need to use this option when a library symbol is redefined. The linker issues error `L2044` if you need to use this option. The minimum abbreviation for this option is **/NOE**.

■ **Syntax**

/NON⟦ULLSDOSSEG⟧

The **/NONULLSDOSSEG** option directs the linker to arrange segments in the same order as they are arranged by the **/DOSSEG** option. The only difference is that the **/DOSSEG** option inserts 16 null bytes at the beginning of the **_TEXT** segment (if it is defined), whereas **/NONULLSDOSSEG** does not insert these extra bytes.

If the linker is given both the **/DOSSEG** and **/NONULLSDOSSEG** options, the **/NONULLSDOSSEG** option will always take precedence. Therefore you can use **/NONULLSDOSSEG** to override the **DOSSEG** comment record commonly found in run-time libraries. The minimum abbreviation for this option is **/NON**.

■ **Syntax**

INC⟦REMENTAL⟧

/PADC⟦ODE⟧:*bytes*

/PADD⟦ATA⟧:*bytes*

The last three options are explained in Section 9 below, "The ILINK Utility."

Section 5
The BIND Utility

The Microsoft Operating System/2 Bind utility (**BIND**) converts protected-mode programs so that they can run in real mode as well as protected mode. Not every protected-mode program can readily be converted. Programs you wish to convert should make no system calls other than calls to the functions listed in the Family API. (The Family API is a subset of the API functions and is summarized in the *Microsoft Operating System/2 Programmer's Reference.*)

The **BIND** utility must "bind" dynamic-link functions; that is, the utility brings an application program together with libraries, and links everything into a single standalone file which can run in real mode. The **BIND** utility also alters the executable-file format of the program, so that it is recognized as a standard executable file by both DOS 3.x and OS/2.

There are three components to the **BIND** utility:

Item	**Description**
BIND	This utility merges the executable file with the appropriate libraries as described above.
loader	This tool loads the OS/2 executable file when running DOS 2.x or 3.x and simulates the OS/2 startup conditions in an environment. The loader consists of code that is stored in **BIND.EXE**, and then copied into files as needed.
API.LIB	This library simulates the OS/2 API in an environment.

5.1 Binding Libraries

The **BIND** utility replaces Family-API calls with simulator routines from the standard (object-code) library **API.LIB**. However, your program may also make dynamic-link calls to functions outside the API (that is, you can make dynamic-link calls that are not system calls). This section explains how **BIND** can accommodate these calls.

If your program makes dynamic-link calls to functions outside the API, use the *linklibs* field described in Section 5.3, "The BIND Command Line." **BIND** searches each of the *linklibs* for object code corresponding to the imported functions. In addition, if you are using import definitions with either the *ordinal* or the *internalname* option, you will need to specify import libraries so that the functions you call can be identified correctly. (For a discussion of various options within import definitions, see Section 7, "Using Module-Definition Files.")

5.2 Binding Functions as Protected Mode Only

If your program freely makes non-Family-API calls without regard to which operating system is in use, then the program cannot be converted for use in real mode. However, you may choose to write a program so that it first checks the operating system, and then restricts system calls (to the Family API) when running in real mode. The **BIND** utility supports conversion of these programs.

By using the **/n** command-line option, described below, you can specify a list of functions supported in protected mode only. If your program ever attempts to call one of these functions when running in real mode, then the **BadDynLink** system function is called and aborts your program. The advantage of this option is that it helps resolve external references. Yet it remains the responsibility of your program to check the operating-system version, and ensure that not one of these functions is ever called in real mode.

If your program makes calls (either directly or indirectly) to non-Family-API system calls, but you do *not* use the **/n** option, then **BIND** will fail to convert your program.

5.3 The BIND Command Line

Invoke **BIND** with the following command line:

BIND *infile* [[*implibs*]] [[*linklibs*]] [[**/o** *outfile*]] [[**/n** **@***file*]] [[**/n** *names*]] [[**/m** *mapfile*]]

The meaning of each command-line field and option is explained below:

The *infile* field contains the name of the OS/2 application. The file name may contain a complete path name. The file extension is optional; if you provide no extension, then .EXE is assumed.

The *implibs* field contains the name of one of more import libraries. As explained above, use this field if your program uses an import definition with either the *ordinal* or *internalname* fields.

Note

If you want to specify a 64-kilobyte (K) default data segment when running in real mode, then specify the file **APILMR.OBJ**, which guarantees a 64K stack. The reason this object file may be necessary is that a protected-mode application is not automatically given a 64K default data segment; a protected-mode application is only allocated the space it specifically requests. If you do not specify the file **APILMR.OBJ**, then you may not have the local heap area you need when you run in real mode.

The *linklibs* field contains the name of one or more standard libraries. Use this field to supply object code needed to resolve dynamic-link calls. If this field is empty, then the library **API.LIB** is automatically included. However, if you specify any libraries, then **API.LIB** is not assumed, and you need to give a complete path name for each library you specify.

The *outfile* is the name of the bound application, and may contain a full path name. The default value of this field is *infile*. (Whatever name is used for the *infile* field also becomes the default for *outfile*.)

The **/n** option provides a way of listing functions that are supported in protected mode only. As explained above, if any of these functions are ever called in real mode, then the **BadDynLink** function will be called to abort the program. The **/n** option can be used either with a list of one or more *names* (separated by spaces), or with a *file* preceded by the @ sign. The *file* should consist of a list of functions, one per line.

The **/m** option causes a link map to be generated for the DOS 3.x environment of the .EXE file. The *mapfile* is the destination of the link map. If no *mapfile* is specified with the **/m** option, then the destination of the link map is standard output.

5.4 BIND Operation

BIND produces a single executable file, which can run on either OS/2 or DOS 3.x. To complete this task, **BIND** executes three major steps:

1. Reads in the dynamic-link entry points (for imported functions) from the OS/2 executable file and outputs to a temporary object file the **EXTDEF** object records for each imported item. Each **EXTDEF** record tells the linker of an external reference that needs to be resolved through ordinary linking.

2. Invokes the linker, giving the executable file, the temporary object file, the **API.LIB** file, and any other libraries specified on the **BIND** command line. The linker produces an executable file which can run in real mode, by linking in the loader and the API-simulator routines.

3. Merges the protected-mode and real-mode executable files, to produce a single file which can run in either mode.

5.5 Executable-File Layout

OS/2 executable files have two headers. The first header has a DOS 3.x format. The second header has the OS/2 format. When the executable file is run on an OS/2 system, it ignores the first header and uses the OS/2 format. When run under DOS 3.x, the old header is used to load the file. Figure 5.1 shows the arrangement of the merged headers.

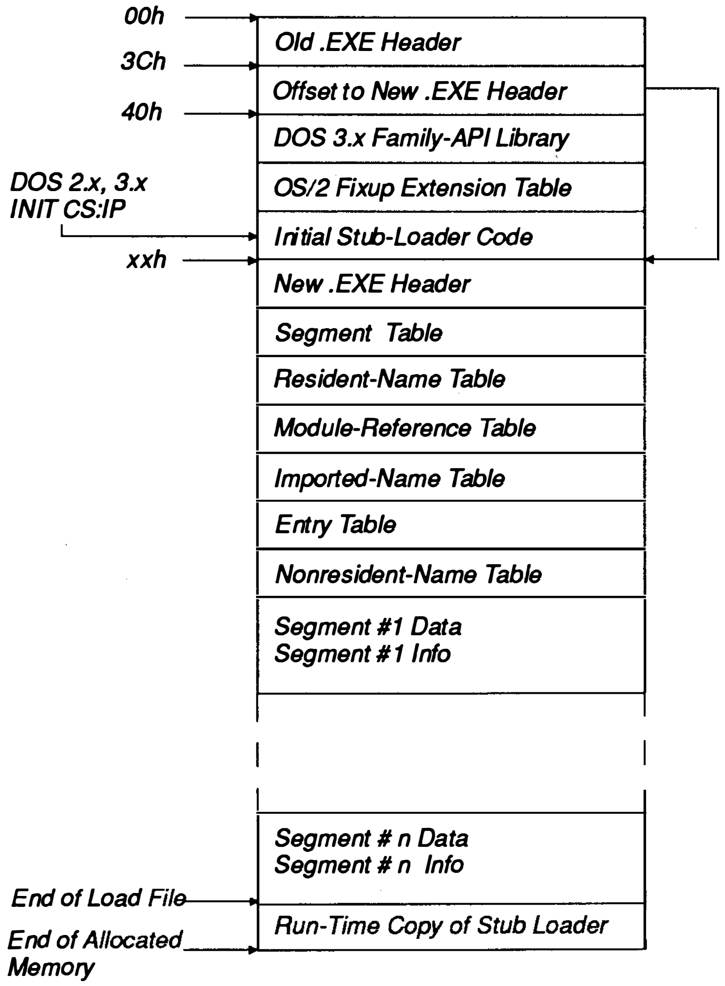

Figure 5.1 OS/2 Executable-File Header

Section 6
The IMPLIB Utility

This section summarizes the use of the Microsoft Import Library Manager utility (**IMPLIB**), and assumes you are familiar with the concepts of import libraries, dynamic linking, and module-definition files. If you are not familiar with these concepts, read Section 3, "About Linking in OS/2."

You can create an import library for use by other programmers in resolving external references to your dynamic-link library. The **IMPLIB** command creates an import library, which is a file with a **.LIB** extension that can be read by the OS/2 linker. The **.LIB** file can be specified in the **LINK** command line with other libraries. Import libraries are recommended for all dynamic-link libraries. Without the use of import libraries, external references to dynamic-link routines must be declared in an **IMPORTS** statement in the module-definition file for the application being linked. **IMPLIB** is supported only in protected mode.

■ Syntax

IMPLIB *implibname mod-def-file* ⟦*mod-def-file...*⟧

The *implibname* is the name you wish the new import library to have.

The *mod-def-file* is the name of a module-definition file for the dynamic-link module. You may enter more than one.

■ Example

The following command creates the import library named **MYLIB.LIB** from the module-definition file **MYLIB.DEF**:

```
IMPLIB mylib.lib mylib.def
```

Section 7
Using Module-Definition Files

A module-definition file describes the name, attributes, exports, imports, and other characteristics of an application or library for OS/2 or Microsoft Windows. This file is required for Windows applications and libraries, and is also required for dynamic-link libraries that run under OS/2.

A module-definition file contains one or more "module statements." Each module statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the number and names of exported and imported functions. The module statements and the attributes they define are listed as follows:

Statement	Attribute
NAME	Names application (no library created)
LIBRARY	Names dynamic-link library (no application created)
DESCRIPTION	Describes the module in one line
CODE	Gives default attributes for code segments
DATA	Gives default attributes for data segments
SEGMENTS	Gives attributes for specific segments
STACKSIZE	Specifies local-stack size, in bytes
EXPORTS	Defines exported functions
IMPORTS	Defines imported functions
STUB	Adds a DOS 3.x executable file to the beginning of the module, usually to terminate the program when run in real mode
HEAPSIZE	Specifies local-heap size, in bytes
PROTMODE	Specifies that the module runs only in DOS protected mode
OLD	Preserves import information from a previous version of the library

REALMODE	Relaxes some restrictions that the linker imposes for protected-mode programs
EXETYPE	Identifies operating system

The following rules govern the use of these statements in a module-definition file:

1. If you use either a **NAME** or a **LIBRARY** statement, it must precede all other statements in the module-definition file.

2. You can include source-level comments in the module-definition file, by beginning a line with a semicolon (;). The OS/2 utilities ignore each such comment line.

3. Module-definition keywords (such as **NAME**, **LIBRARY**, and **SEGMENTS**) must be entered in uppercase letters.

The following sample module-definition file gives module definitions for a dynamic-link library. This sample file includes one source-level comment and four statements.

```
; Sample module-definition file

LIBRARY

DESCRIPTION 'Sample .DEF file for a dynamic-link library'

CODE       PRELOAD

STACKSIZE  1024

EXPORTS
    Init   @1
    Begin  @2
    Finish @3
    Load   @4
    Print  @5
```

The meaning of each of these fields is explained in the sections that follow, which describe module-definition statements, and give syntax and examples.

7.1 The NAME Statement

The **NAME** statement identifies the executable file as an application and optionally defines the name.

■ Syntax

NAME[[*appname*]] [[*apptype*]]

■ Remarks

If an *appname* is given, it becomes the name of the application as it is known by OS/2. This name can be any valid file name. If no *appname* is given, the name of the executable file—with the extension removed—becomes the name of the application.

The *apptype* field is used by a future version of OS/2, and should be declared for compatibility with this future version.

If *apptype* is given, it defines the type of application being linked. This information is kept in the executable-file header. You do not need to use this field unless you may be using your application in a Windows environment. The *apptype* field may have one of the following values:

Keyword	Meaning
WINDOWAPI	Real-mode Windows application. The application uses the API provided by Windows and must be executed in the Windows environment.
WINDOWCOMPAT	Windows-compatible application. The application can run inside Windows, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions which are supported in Windows applications.
NOTWINDOWCOMPAT	Application is not Windows compatible and must operate in a separate screen group from Windows.

If the **NAME** statement is included in the module-definition file, then the **LIBRARY** statement cannot appear. If neither a **NAME** statement nor a **LIBRARY** statement appears in a module-definition file, the default is **NAME**; that is, the linker acts as though a **NAME** statement were included, and thus creates an application rather than a library.

■ Example

The following example assigns the name `calendar` to the application being defined:

```
NAME calendar WINDOWCOMPAT
```

7.2 The LIBRARY Statement

The **LIBRARY** statement identifies the executable file as a dynamic-link library, and it can specify the name of the library or the type of library-module initialization required.

■ Syntax

LIBRARY [[*libraryname*]] [[*initialization*]]

■ Remarks

If a *libraryname* is given, it becomes the name of the library as it is known by OS/2. This name can be any valid file name. If no *libraryname* is given, the name of the executable file—with the extension removed—becomes the name of the library.

The *initialization* field is optional and can have one of the two values listed below. If neither is given, then the *initialization* default is **INITGLOBAL.**

Keyword	Meaning
INITGLOBAL	The library-initialization routine is called only when the library module is initially loaded into memory.
INITINSTANCE	The library-initialization routine is called each time a new process gains access to the library.

If the **LIBRARY** statement is included in a module-definition file, then the **NAME** statement cannot appear. If no **LIBRARY** statement appears in a module-definition file, the linker assumes that the module-definition file is defining an application.

■ Example

The following example assigns the name `calendar` to the dynamic-link module being defined, and specifies that library initialization is performed each time a new process gains access to `calendar`.

```
LIBRARY calendar INITINSTANCE
```

7.3 The DESCRIPTION Statement

The **DESCRIPTION** statement inserts the specified *text* into the application or library. This statement is useful for embedding source-control or copyright information into an application or library.

■ Syntax

DESCRIPTION *'text'*

■ Remarks

The *text* is a one-line string enclosed in single quotation marks. Use of the **DESCRIPTION** statement is different from the inclusion of a comment, because comments—lines that begin with a semicolon (;)—are not placed in the application or library.

■ Example

The following example inserts the text `Template Program` into the application or library being defined:

```
DESCRIPTION 'Template Program'
```

7.4 The CODE Statement

The **CODE** statement defines the default attributes for code segments within the application or library.

■ Syntax

CODE[[*attribute...*]]

■ Remarks

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last. The last three fields have no effect on OS/2 code segments and are included for use with Microsoft Windows.

Field	Values
load	**PRELOAD, LOADONCALL**
executeonly	**EXECUTEONLY, EXECUTEREAD**
iopl	**IOPL, NOIOPL**
conforming	**CONFORMING, NONCONFORMING**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**NONDISCARDABLE, DISCARDABLE**

The *load* field determines when a code segment is to be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded automatically, at the beginning of the program.
LOADONCALL	The segment is not loaded until accessed (the default).

The *executeonly* field determines whether a code segment can be read as well as executed. This field contains one of the following keywords:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed.
EXECUTEREAD	The segment can be both executed and read (the default).

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The code segment has I/O privilege.
NOIOPL	The code segment does not have I/O privilege (the default).

The *conforming* field specifies whether or not a code segment is a 286 "conforming" segment. The concept of a conforming segment deals with privilege level (the range of instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. This field contains one of the following keywords:

Keyword	Meaning
CONFORMING	The segment is conforming.
NONCONFORMING	The segment is nonconforming (the default).

The *shared* field determines whether all instances of the program can share a given code segment. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all code segments are shared. The *shared* field contains one of the following keywords: **SHARED** or **NONSHARED** (the default).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The *discard* field determines whether a segment can be swapped out to disk by the operating system when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *shared* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

■ Example

The following example sets defaults for the module's code segments, so that they are not loaded until accessed and so that they have I/O hardware privilege:

```
CODE LOADONCALL IOPL
```

7.5 The DATA Statement

The **DATA** statement defines the default attributes for the data segments within the application or module.

■ Syntax

DATA ⟦*attribute...*⟧

■ Remarks

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are present below, along with legal values. In each case, the default value is listed last. The last two fields have no effect on OS/2 data segments, but are included for use with Microsoft Windows.

Field	Values
load	**PRELOAD, LOADONCALL**
readonly	**READONLY, READWRITE**
instance	**NONE, SINGLE, MULTIPLE**
iopl	**IOPL, NOIPL**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**DISCARDABLE, NONDISCARDABLE**

The *load* field determines when a segment will be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded when the program begins execution.
LOADONCALL	The segment is not loaded until it is accessed (the default).

The *readonly* field determines the access rights to a data segment. This field contains one of the following keywords:

Keyword	Meaning
READONLY	The segment can only be read.
READWRITE	The segment can be both read and written to (the default).

The *instance* field affects the sharing attributes of the automatic data segment, which is the physical segment represented by the group name **DGROUP**. (This segment group makes up the physical segment which contains the local stack and heap of the application.) The *instance* field contains one of the following keywords:

Keyword	Meaning
NONE	No automatic data segment is created.
SINGLE	A single automatic data segment is shared by all instances of the module. In this case, the module is said to have "solo" data. This keyword is the default for dynamic-link libraries.
MULTIPLE	The automatic data segment is copied for each instance of the module. In this case, the module is said to have "instance" data. This keyword is the default for applications.

The *iopl* field determines whether or not data segments have I/O privilege (that is, whether or not they can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The data segments have I/O privilege.
NOIOPL	The data segments do not have I/O privilege (the default).

The *shared* field determines whether all instances of the program can share a **READ-WRITE** data segment. Under OS/2, this field is ignored by the linker if the segment has the attribute **READONLY**, since **READONLY** data segments are always shared. The *shared* field contains one of the following keywords:

Keyword	Meaning
SHARED	One copy of the data segment will be loaded and shared among all processes accessing the module.
NONSHARED	The segment cannot be shared, and must be loaded separately for each process (the default).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system, when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *discard* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

Note

The linker makes the automatic data segment attribute (specified by an *instance* value of **SINGLE** or **MULTIPLE**) match the sharing attribute of the automatic data segment (specified by a *shared* value of **SHARED** or **NONSHARED**). Solo data (specified by **SINGLE**) force shared data segments by default. Instance data (specified by **MULTIPLE**) force nonshared data by default. Similarly, **SHARED** forces solo data, and **NONSHARED** forces instance data.

If you give a contradictory **DATA** statement (e.g., `DATA SINGLE NONSHARED`), all segments in **DGROUP** are shared, and all other data segments are nonshared by default. If a segment that is a member of **DGROUP** is defined with a *sharing* attribute that conflicts with the automatic data type, a warning about the bad segment is issued, and the segment's flags are converted to a consistent sharing attribute. For example, the following

```
DATA SINGLE
SEGMENTS
_DATA CLASS 'DATA' NONSHARED
```

is converted to

```
_DATA CLASS 'DATA' SHARED
```

■ **Example**

The following example defines the application's data segment so that it is loaded only when it is accessed and so that it cannot be shared by more than one copy of the program:

```
DATA LOADONCALL NONSHARED
```

By default, the data segment can be read and written, the automatic data segment is copied for each instance of the module, and the data segment has no I/O privilege.

7.6 The SEGMENTS Statement

The **SEGMENTS** statement defines the attributes of one or more segments in the application or library on a segment-by-segment basis. The attributes specified by this statement override defaults set in **CODE** and **DATA** statements.

■ Syntax

SEGMENTS
segmentdefinitions

■ Remarks

The **SEGMENTS** keyword marks the beginning of the segment definitions. This keyword can be followed by one or more segment definitions, each on a separate line (limited by the number set by the linker's **/SEGMENTS** option, or 128 if the option is not used). The syntax for each segment definition is as follows:

segmentname ⟦**CLASS** '*classname*'⟧⟦*attribute...*⟧

Each segment definition begins with a *segmentname*, which can be placed in optional single quotation marks ('). The quotation marks are required if *segmentname* conflicts with a module-definition keyword, such as **CODE** or **DATA**.

The **CLASS** keyword specifies the class of the segment. The single quotation marks (') are required around *classname*. If you do not use the **CLASS** argument, the linker assumes that the class is **CODE**.

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last.

Field	Values
load	**PRELOAD, LOADONCALL**
readonly	**READONLY, READWRITE**
executeonly	**EXECUTEONLY, EXECUTEREAD**
iopl	**IOPL, NOIOPL**
conforming	**CONFORMING, NONCONFORMING**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**DISCARDABLE, NONDISCARDABLE**

The *load* field determines when a segment is to be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded automatically, at the beginning of the program.
LOADONCALL	The segment is not loaded until accessed (the default).

The *readonly* field determines the access rights to a data segment. This field contains one of the following keywords:

Keyword	Meaning
READONLY	The segment can be read only.
READWRITE	The segment can be both read and written to (the default).

The *executeonly* field determines whether a code segment can be read as well as executed. (The attribute has no effect on data segments.) This field contains one of the following keywords:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed.
EXECUTEREAD	The segment can be both executed and read (the default).

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The segments have I/O privilege.
NOIOPL	The segments do not have I/O privilege (the default).

The *conforming* field specifies whether or not a code segment is a 286 "conforming" segment. The concept of a conforming segment deals with privilege level (the range of instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. (The attribute has no effect on data segments.) This field contains one of the following keywords:

Keyword	Meaning
CONFORMING	The segment is conforming.
NONCONFORMING	The segment is nonconforming (the default).

The *shared* field determines whether all instances of the program can share a **READ-WRITE** segment. For code segments and **READONLY** data segments, this field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all code segments and all **READONLY** data segments are shared. The *shared* field contains one of the following keywords: **SHARED** or **NONSHARED** (the default).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system, when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *shared* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

■ Example

The following example specifies segments named `cseg1`, `cseg2`, and `dseg`. The first segment is assigned class `mycode` and the second is assigned **CODE**. Each segment is given different attributes.

```
SEGMENTS
    cseg1 CLASS 'mycode' IOPL
    cseg2 EXECUTEONLY PRELOAD CONFORMING
    dseg  CLASS 'data' LOADONCALL READONLY
```

7.7 The STACKSIZE Statement

The **STACKSIZE** statement performs the same function as the **/STACKSIZE** linker option. It overrides the size of any stack segment defined in an application. (The **STACKSIZE** statement overrides the **/STACKSIZE** option).

■ Syntax

STACKSIZE *number*

■ Remarks

The *number* must be an integer. The *number* is considered to be in decimal format by default, but you can use C notation to specify hexadecimal or octal.

■ Example

The following example allocates 4096 bytes of local-stack space:

```
STACKSIZE 4096
```

7.8 The EXPORTS Statement

The **EXPORTS** statement defines the names and attributes of the functions exported to other modules, and of the functions that run with I/O privilege. The term "export" refers to the process of making a function available to other run-time modules. By default, functions are hidden from other modules at run time.

■ Syntax

EXPORTS
    exportdefinitions

■ Remarks

The **EXPORTS** keyword marks the beginning of the export definitions. It may be followed by up to 3072 export definitions, each on a separate line. You need to give an export definition for each dynamic-link routine that you want to make available to other modules. The syntax for an export definition is as follows:

entryname⟦*=internalname*⟧ ⟦*@ord*⟦**RESIDENTNAME**⟧⟧ ⟦*pwords*⟧ ⟦**NODATA**⟧

The *entryname* specification defines the function name as it is known to other modules. The optional *internalname* defines the actual name of the export function as it appears within the module itself; by default, this name is the same as *entryname.*

The optional *ord* field defines the function's ordinal position within the module-definition table. If this field is used, then the function's entry point can be invoked by name or by ordinal. Use of ordinal positions is faster and may save space.

The optional keyword **RESIDENTNAME** specifies that the function's name be kept resident in memory at all times. This keyword is applicable only if the *ord* option is used, because if the *ord* option is not used, OS/2 automatically keeps the names of all exported functions resident in memory anyway.

The *pwords* field specifies the total size of the function's parameters, as measured in words (the total number of bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults the *pwords* field to determine how many words to copy from the caller's stack to the I/O-privileged function's stack.

The optional keyword **NODATA** is ignored by OS/2, but is provided for use by real-mode Windows.

Normally, the **EXPORTS** statement is only meaningful for functions within dynamic-link libraries, and for functions which execute with I/O privilege.

■ Example

The following **EXPORTS** statement defines three export functions: `SampleRead`, `StringIn`, and `CharTest`. The first two functions can be accessed either by their exported names or by an ordinal number. Note that in the module's own source code, these functions are actually defined as `read2bin` and `str1`, respectively. The last function runs with I/O privilege, and therefore is given with the total size of the parameters: six words.

```
EXPORTS
       SampleRead = read2bin  @8
       StringIn = str1        @4  RESIDENTNAME
       CharTest   6
```

7.9 The IMPORTS Statement

The **IMPORTS** statement defines the names of the functions that will be imported for the application or library. The term "import" refers to the process of declaring that a symbol is defined in another run-time module (a dynamic-link library). Typically, **LINK** uses an import library (created by the **IMPLIB** utility) to resolve external references to dynamic-link symbols. However, the **IMPORTS** statement provides an alternative for resolving these references within a module.

■ Syntax

IMPORTS
   importdefinitions

■ Remarks

The **IMPORTS** keyword marks the beginning of the import definitions. This keyword is followed by one or more import definitions, each on a separate line. The only limit on the number of import definitions is that the total amount of space required for their names must be less than 64K. Each import definition corresponds to a particular function. The syntax for an import definition is as follows:

[[*internalname*=]]*modulename.entry*

The *internalname* specifies the name that the importing module actually uses to call the function. Thus, *internalname* will appear in the source code of the importing module, though the function may have a different name in the module where it is defined. By default, *internalname* is the same as the name given in *entry*.

The *modulename* is the name of the application or library that contains the function.

The *entry* field determines the function to be imported, and can be a name or an ordinal value. (Ordinal values are set in an **EXPORTS** statement.) If an ordinal value is given, then the *internalname* field is required.

Note

> A given function has a name for each of three different contexts. The function has a name used by the exporting module (where it is defined), a name used as an entry point between modules, and a name as it is used by the importing module (where it is called). If neither module uses the optional *internalname* field, then the function will have the same name in all three contexts. If either of the modules use the *internalname* field, then the function may have more than one distinct name.

■ Example

The following **IMPORTS** statement defines three functions to be imported: `SampleRead`, `SampleWrite`, and a function that has been assigned an ordinal value of 1. The functions are found in the modules `Sample`, `SampleA`, and `Read`, respectively. The function from the `Read` module is referred to as `ReadChar` in the importing module; the original name of the function, as it is defined in the `Read` module, may or may not be known.

```
IMPORTS
    Sample.SampleRead
    SampleA.SampleWrite
    ReadChar = Read.1
```

7.10 The STUB Statement

The **STUB** statement adds *filename*, a DOS 3.x executable file, to the beginning of the application or library being created. The stub will be invoked whenever the module is executed under DOS 2.x or DOS 3.x. Typically, the stub displays a message and terminates execution. (By default, the linker adds its own standard stub for this purpose.)

■ Syntax

STUB *'filename'*

■ Remarks

If the linker does not find this file in the current directory, it searches in the list of directories specified in the **PATH** environment variable.

■ Example

The following example appends the DOS executable file *sample.exe* to the beginning of the module:

```
STUB 'STOPIT.EXE'
```

The file **STOPIT.EXE** is executed when you attempt to run the module under DOS.

7.11 The HEAPSIZE Statement

The **HEAPSIZE** statement defines the size of the application's local heap, in bytes. This value affects the size of the automatic data segment.

■ Syntax

HEAPSIZE *bytes*

■ Remarks

The *bytes* field is an integer number, which is considered decimal by default. However, hexadecimal and octal numbers can be entered by using C notation.

■ **Example**

```
HEAPSIZE 4000
```

7.12 The PROTMODE Statement

The **PROTMODE** statement specifies that the module will run only in protected mode and not in Windows or dual mode. This statement is always optional, and permits a protected-mode-only application to omit some information from the executable-file header.

■ **Syntax**

PROTMODE

■ **Remarks**

If this statement is not included in the module-definition file, the linker assumes that the application can be run in either real or protected mode.

7.13 The OLD Statement

The **OLD** statement directs the linker to search another dynamic-link module for export ordinals. For more information on ordinals, consult the sections above on the **EXPORTS** and **IMPORTS** statements. Exported names in the current module that match exported names in the **OLD** module are assigned ordinal values from that module unless one of the following conditions is in effect: the name in the **OLD** module has no ordinal value assigned, or an ordinal value is explicitly assigned in the current module.

■ **Syntax**

OLD *'filename'*

■ **Remarks**

This statement is useful for preserving export ordinal values, throughout successive versions of a dynamic-link module. The **OLD** has no effect on application modules.

7.14 The REALMODE Statement

The **REALMODE** statement is analogous to the **PROTMODE** statement, and is provided for use with real-mode Windows applications.

■ Syntax

REALMODE

■ Remarks

REALMODE specifies that the application runs only in real mode. With this statement, the linker relaxes some of the restrictions that it imposes on programs running in protected mode.

7.15 The EXETYPE Statement

The **EXETYPE** statement specifies in which operating system the application (or dynamic-link library) is to run. This statement is optional and provides an additional degree of protection against the program being run in an incorrect operating system.

■ Syntax

EXETYPE [[**OS2** | **WINDOWS** | **DOS4**]]

■ Remarks

The **EXETYPE** keyword must be followed by a des of the operating system, either **OS2** (for OS/2 applications and dynamic-link libraries), **WINDOWS**, or **DOS4**. If no **EXETYPE** statement is given, then **EXETYPE OS2** is assumed by an operating system that is loading the program.

The effect of **EXETYPE** is simply to set bits in the header which identify operating system type. Operating system loaders may check these bits.

Section 8
Using the /X Option with MAKE

In addition to the options listed in Section 14.5 of the Microsoft CodeView and Utilities manual, "Specifying MAKE Options," the version of the Microsoft Program Maintenance Utility (**MAKE**) that accompanies Microsoft OS/2 has an additional option, which redirects error output. This option is particularly valuable if you run **MAKE** from a batch file, and you want to collect any error messages that occur.

■ Syntax

/X*file*

When you specify the **/X** option on the **MAKE** command line, then the **MAKE** utility will send all error output to *file*, which can be either a file or device. If **MAKE** cannot redirect output to *file*, then it will issue the following fatal error message:

```
U1015: file : error redirection failed
```

For example, **MAKE** issues the message shown above when you try to redirect error output to a read-only file on a DOS 3.x network.

In the discussion above, "error output" is defined as output which is written to standard error output. The file handle for standard error output is usually abbreviated as **stderr** in C programs.

By default, **MAKE** error messages are always sent to **stderr**.

Section 9
The ILINK Utility

The Microsoft Incremental Linker (**ILINK**) is a utility that can enable you to link your OS/2 or Windows application much faster. (It cannot work with DOS applications other than Windows.) You can benefit from its use when you change a small subset of the modules used to link a program. The program can use any memory model, but **ILINK** is most effective with large- and medium-memory-model programs. Furthermore, to benefit from **ILINK** you need to follow certain restrictions that are described in this chapter. Should **ILINK** fail to link your changes into the executable file, it will automatically invoke the full linker, **LINK.** You must first run the full linker with certain new options, described below, before you can use **ILINK.**

Note

You can use **ILINK** to develop dynamic-link libraries as well as applications. Everything said in this chapter about applications and executable files applies to dynamic-link libraries as well. This chapter uses the term "library" to refer specifically to an object-code library (a **.LIB** file).

This chapter covers the following topics:

- Definitions
- Guidelines for using **ILINK**
- The development process
- Running **ILINK**
- How **ILINK** works
- Incremental violations

9.1 Definitions

Incremental linking involves certain specialized concepts. You may need to review the following list of terms in order to understand the rest of this chapter:

Term	Meaning
segment	A contiguous area of memory up to 64K in size. See the definitions of "physical segment" and "logical segment" below.
module	A unit of code or data defined by one source file. In BASIC, Pascal, and large-memory-model C and FORTRAN programs, each module corresponds to a different segment. In small-memory-model programs, all code modules contribute to one code segment, and all data modules contribute to one data segment.
memory model	The memory model determines the number of code and data segments in a program. BASIC programs are always large memory model.
physical segment	A segment listed in the executable file's segment table. Each physical segment has a distinct segment address, whereas logical segments may share a segment address. A physical segment usually contains one logical segment, but it can contain more than one.
logical segment	A segment defined in an object module. Each physical segment other than **DGROUP** contains exactly one logical segment, except when you use the **GROUP** directive in a **MASM** module. (Linking with the **/PACKCODE** option can also create more than one logical segment per physical segment.)
code symbol	The address of a function, subroutine, or procedure.
data symbol	The address of a global or static data object. The concept of data symbol includes all data objects except local (stack-allocated) or dynamically allocated data.

9.2 Guidelines for Using ILINK

The incremental linker, **ILINK**, works much faster than the full linker because **ILINK** replaces only those modules which have changed since the last linking. It avoids much of the work done by **LINK**.

To enable incremental linking, you need to follow four major guidelines. If your changes exceed the scope allowed by these guidelines, then a full link is necessary.

1. Do not alter any **.LIB** files that you are using to create the executable file.

2. Put padding at the end of data and small-memory-model code modules, by using the **/PADCODE** and **/PADDATA** options presented in Section 9.3, "The Development Process."

   By putting padding at the end of a module, you enable the module to grow without forcing a full relinking. However, if the module is the only module contributing to its physical segment, then padding is not necessary.

   In practice this means that you can avoid padding if you have a BASIC, Pascal, FORTRAN, or C program (other than a small-memory-model C program), you do not call a **MASM** module that uses the **GROUP** directive, and you do not add to the size of the default data segment; consult your language documentation for information about what is placed in this area.

3. Do not delete code symbols (functions and procedures) that are referenced by another module. You can, however, move or add to these symbols.

4. Do not move or delete data symbols (global data). You can add data symbols at the end of your data definitions, but you cannot add new communal data symbols (for example, C uninitialized variables or BASIC **COMMON** statements).

9.3 The Development Process

To develop a software project with **ILINK**, follow the steps listed below:

1. Use the full linker during early stages of developing your application or dynamic-link library. You will not be ready to take advantage of **ILINK** until you have a number of different code and data segments present.

2. Prepare for incremental linking by using the special linker options described below.

3. Incrementally link with **ILINK** after any small changes are made.

4. Relink with **LINK** after any major changes are made (for example, if you want to add an entirely new module, you want to greatly expand one of the segments or modules, or you want to redefine symbols that are shared between segments).

5. Repeat steps 3 and 4 as necessary.

Three options, **/INCREMENTAL**, **/PADCODE**, and **/PADDATA**, have been added to **LINK** to allow the use of **ILINK**. Here is an example of how they might appear on the command line:

```
LINK /INCREMENTAL /PADDATA:16 /PADCODE:256 @PROJNAME.LNK
```

Sections 9.3.1–9.3.3 present the new options.

9.3.1 The /INCREMENTAL Option

■ **Syntax**

/INC[[**REMENTAL**]]

The **/INCREMENTAL** option must be used with the full linker (**LINK**) in order to prepare for subsequent linking with **ILINK**. The use of this option produces a **.SYM** file and a **.ILK** file, which contain extra information needed by **ILINK**. Note that this option is not compatible with **/EXEPACK**.

9.3.2 The /PADCODE Option

■ **Syntax**

/PADC[[**ODE**]]:*padsize*

The **/PADCODE** option causes **LINK** to add filler bytes to the end of each code module. The option is followed by a colon and the number of bytes to add. (Decimal radix is assumed, but you can specify special octal or hexadecimal numbers by using a C-language prefix.) Thus

```
/PADCODE:256
```

adds an additional 256 bytes to each module. The default size for code-module padding is 0 bytes.

Note

Code padding is usually not necessary for large- and medium-memory-model programs, but is recommended for small-, compact-, and mixed-memory-model programs, and for **MASM** programs in which code segments are grouped.

To be recognized as a code segment, a segment must be declared with class name `'CODE'`. (Microsoft high-level languages automatically use this declaration for code segments.)

9.3.3 The /PADDATA Option

■ **Syntax**

/PADD[[ATA]]:*padsize*

The **/PADDATA** option performs a function similar to **/PADCODE**, except that it specifies padding for data segments (or data modules, if the program uses small or medium memory model). Thus

```
/PADDATA:32
```

adds an additional 32 bytes to each module. The default size for data-segment padding is 16 bytes.

Note

If you specify too large a value for *padsize*, you may exceed the 64K limitation on the size of the default data segment.

9.4 Running ILINK

You can attempt to link the project with **ILINK** at any time after the project has been linked with the **/INCREMENTAL** option. The following two sections discuss the files needed for linking with **ILINK** and the command required to invoke **ILINK**.

9.4.1 Files Required for Using ILINK

The files that are required for linking using **ILINK** are **ILINK.EXE, EXEC.EXE,** and your project files, which include:

1. *projname*.EXE (the file to incrementally link)
2. *projname*.SYM (the symbol file)
3. *projname*.ILK (the **ILINK** support file)
4. ***.OBJ** (the changed **.OBJ** files)

It is strongly suggested that you place **EXEC.EXE** in a directory listed in the **PATH** environment variable.

9.4.2 The ILINK Command Line

The syntax for the **ILINK** command line is shown below. **ILINK** options are not case sensitive.

ILINK [[/**a**]] [[/**c**]] [[/**v**]] [[/**i**]] [[/**e** "*commands*"]] *projname*[[*modulelist*]]

The /**a** option specifies that all object files are to be checked to see if they have changed since the last linking process. This is done by comparing the dates and times of all **.OBJ** files with those of the executable file. An attempt is made to incrementally link all of the files that have changed.

The /**c** option specifies case sensitivity for all public symbol names.

The /**v** option specifies verbose mode, and directs **ILINK** to display more information. Specifically, when in verbose mode, **ILINK** lists the modules that have changed.

The /**i** option specifies that only an incremental link is to be attempted; if the incremental link fails, a full link is not performed.

The /**e** option specifies a list of commands to be executed if the incremental link fails. The commands are enclosed in double quotes, and if more than one command is given, they must be separated by spaces and a semicolon. The characters `%s` are replaced by *projname* when the command is carried out. In the following example, if the incremental link fails, then **ILINK** carries out the commands `link myproj.obj` and `rc myproj.exe`:

```
ILINK /e "link @%s.obj ; rc %s.exe" myproj
```

The *projname* field contains the name of the executable file that is to be incrementally linked.

You can use the optional *modulelist* field to list all the modules that have changed. (No extensions are required.) This field is an alternative to using the **/a** flag.

■ Examples

Two examples using **ILINK** are shown below. In the first example, the altered modules (`input`, `sort`, and `output`) are explicitly listed on the command line. In the second example, the **-a** option directs **ILINK** to scan all files in the project, in order to discover which modules have changed. The second example also lists commands to be executed in the case that incremental linking fails.

```
ILINK /i wizard input sort output

ILINK /a /e "link @%s.lnk ; rc %s.exe" wizard
```

9.5 How ILINK Works

Instead of searching for records and resolving external references for the entire program, **ILINK** carries out the following operations:

1. **ILINK** replaces the portion of each module that has changed since the last linking (incremental or full linking).

2. **ILINK** alters relocation-table entries for any far (segmented) code symbols that have moved within a segment. (For each reference to a far code symbol, such as a far function call, there is an entry in the relocation table in the executable file's header. Unlike the relocation table of a DOS application, the relocation table of an OS/2 application contains full addresses, not just segment addresses. Thus, by fixing relocation table entries for a code symbol, **ILINK** ensures that all references to the symbol will be correct.)

ILINK makes no modification to modules that have not been changed since the last linking.

9.6 Incremental Violations

There are two kinds of **ILINK** failures: real errors and incremental violations. Real errors are errors that will not be resolved by a full link, such as undefined symbols or invalid **.OBJ** files. If **ILINK** detects a real error, it will display `ERROR` with an explanation, and return a nonzero error code to the operating system. On the other hand, incremental violations consist of changes that are beyond the scope of incremental linking, but can generally be resolved by full linking.

The Microsoft CodeView and Utilities manual explains real errors in detail. The rest of this section describes incremental violations.

9.6.1 Changing Libraries

An incremental violation occurs when a library changes. Furthermore, if an altered module shares a code segment with a library, then **ILINK** will need access to the library as well as to the new module.

Note

If you add a function, procedure, or subroutine call to a library that has never been called before, then **ILINK** will fail with an undefined-symbol error. Performing a full link should resolve this problem.

9.6.2 Exceeding Code/Data Padding

An incremental violation will occur if two or more modules contribute to the same physical segment, and either module exceeds its padding. As discussed in Section 9.3, "The Development Process," padding is the process of adding filler bytes to the end of a module. The filler bytes serve as a buffer zone whenever the module grows in size (that is, whenever the new version of the module is larger than the old).

9.6.3 Moving/Deleting Data Symbols

An incremental violation occurs if a data symbol is moved or deleted. To add new data symbols without requiring a full link, add the new symbols at the end of all other data symbols in the module.

9.6.4 Deleting Code Symbols

You can move or add code symbols, but an incremental violation occurs if you delete any code symbols from a module. Code symbols can be moved within a module but cannot be moved between modules.

9.6.5 Changing Segment Definitions

An incremental violation will result if you add, delete, or change the order of segment definitions. If you are programming in **MASM**, an incremental violation will also result if you alter any **GROUP** directives.

If you are programming with a high-level language, then you need only remember not to add or delete modules between incremental links.

9.6.6 Adding CodeView Debugger Information

If you included CodeView-debugger information for a module the last time you ran a full link (by compiling and linking with CodeView-debugger support), then **ILINK** fully supports CodeView-debugger information for the module. **ILINK** will maintain symbolic information for current symbols, and it will add information for any new symbols. However, if you include CodeView-debugger information for a module which previously did *not* have CodeView-debugger support, an incremental violation will result.

Section 10
The EXEHDR Utility

The Microsoft Segmented EXE File Header Utility (**EXEHDR**) displays the contents of an executable-file header. You can use **EXEHDR** with OS/2 or Windows, and you can use it with an application or dynamic-link library. So there are really four possibilities total. (With a Windows file, some of the meanings of the executable-file-header fields change; consult your Windows documentation for more information.) The principal uses of **EXEHDR** include the following:

- Determining whether a file is an application or a dynamic-link library
- Viewing the attributes set by the module-definition file
- Viewing the number and size of code and data segments

Except where noted otherwise, the specialized terms and keywords mentioned in this section are explained in Section 7, "Using Module-Definition Files."

10.1 The EXEHDR Command Line

To invoke the **EXEHDR** utility, use the following syntax:

EXEHDR [[/v]] *file*

in which *file* is an application or dynamic-link library, for either the OS/2 or Windows environment. The /v option specifies verbose mode, which is discussed in Section 10.3.

Section 10.2 presents sample output and then explains the meaning of each field of the output. Section 10.3 describes additional output that **EXEHDR** produces when it is run in verbose mode.

10.2 EXEHDR Output

This section discusses the meaning of each field in the output below—output produced when `EXEHDR LINK.EXE` is specified on the OS/2 command line. The first six fields list the contents of the segmented-executable-file header. The rest of the output lists each physical segment in the file. (The term "physical segment" is defined in Section 9, "The ILINK Utility.")

```
Module:                    LINK
Description:               Microsoft Segmented-Executable Linker
Data:                      NONSHARED
Initial CS:IP:             seg   2 offset 3d9c
Initial SS:SP:             seg   4 offset 8e40
DGROUP:                    seg   4

no. type address  file  mem   flags
  1 CODE 00003400 0f208 0f208
  2 CODE 00012e00 05b04 05b04
  3 DATA 00018c00 01c1f 01c1f
  4 DATA 0001aa00 01b10 08e40
```

The `Module` field is the name of the application as specified in the **NAME** statement of the module-definition file. If no module definition was used to create the executable file, then this field displays the name assumed by default. If a module definition was used to create the file, but the **LIBRARY** statement appeared instead of the **NAME** statement (thus specifying a dynamic-link library), then the name of the library is given and **EXEHDR** uses the word `Library` instead of `Module`.

The `Description` field gives the contents, if any, of the **DESCRIPTION** statement of the module-definition file used to create the file being examined.

The `Data` field indicates the type of the program's automatic data segment: **SHARED, NONSHARED**, or **NONE**. This type can be specified in a module-definition file, but by default is **NONSHARED** for applications and **SHARED** for dynamic-link libraries. In this context, **SHARED** and **NONSHARED** are equivalent to the **SINGLE** and **MULTIPLE** attributes listed in Section 7.5. (The "automatic data segment" is the physical segment corresponding to the group named **DGROUP**.)

The `Initial CS:IP` field is the program starting address (if an application is being examined) or address of the initialization routine (if a dynamic-link library is being examined).

The `Initial SS:SP` field gives the value of the initial stack pointer.

The `DGROUP` field is the segment number of the automatic data segment. This segment corresponds to the group named **DGROUP** in the program's object modules. Note that segment numbers start with the number 1.

After the contents of the OS/2 executable header is displayed, the contents of the segment table is listed. The following list describes the meaning of each heading in the table. Note that all values are given in hexadecimal radix except for the segment index number.

Heading	Meaning
`no.`	Segment index number, starting with 1, in decimal radix.
`type`	Identification of the segment as a code or data segment. A code segment is any segment with class name ending in `'CODE'`. All other segments are data segments.
`address`	Location within the file, of the contents of the segment.
`file`	Size in bytes of the segment, as contained in the file.
`mem`	Size in bytes of the segment as it will be stored in memory. If the value of this field is greater than the value of the `file` field, then at load time OS/2 pads the additional space with zero values.
`flags`	Segment attributes, as defined in Section 7, "Using Module-Definition Files." If the /v option is not used, then only non-default attributes are listed. Attributes are given in the form specified in Section 7: **READWRITE, PRELOAD**, and so forth. Attributes that are meaningful to Windows but not to OS/2 are displayed as lowercase and in parentheses, (e.g., (`movable`)).

10.3 Output in Verbose Mode

When you specify the /v mode, the **EXEHDR** utility gives all the information discussed in Section 10.2, as well as some additional information. Specifically, when running in verbose mode **EXEHDR** displays the following information in this order:

1. DOS 3.x header information. All OS/2 executable files have a DOS 3.x header, whether bound or not. If the program is not bound, then the DOS 3.x portion consists of a stub that simply terminates the program. For a description of DOS executable-header fields, see the *Microsoft MS-DOS Programmer's Reference,* Chapter 5, or see the chapter on the Microsoft EXE File Header Utility (**EXEMOD**) in the Microsoft CodeView and Utilities manual.

2. OS/2-specific header fields. This output includes the fields described in Section 10.2, except for the segment table. (The segment-table display for verbose mode is described below.)

3. File addresses and lengths of the various tables in the executable file, as in the following example:

```
Resource Table:            00003273 length 0000 (0)
Resident Names Table:      00003273 length 0008 (8)
Module Reference Table:    0000327b length 0006 (6)
Imported Names Table:      00003281 length 0033 (51)
Entry Table:               000032b4 length 0002 (2)
Non-resident Names Table:  000032b6 length 0029 (41)
Movable entry points:      0
Segment sector size:       512
```

The first field in each row gives the name of the table, the second field gives the address of the table within the file, the third field gives the length of the table in hexadecimal radix, and the last field gives the length of the table in decimal radix. See the *Microsoft Operating System/2 Programmer's Reference* for an explanation of each table.

4. Segment table with complete attributes, not just the nondefault attributes. In addition to the attributes described in Section 7, verbose mode also displays two additional attributes:

   The `relocs` attribute is displayed for each segment that has address relocations. Relocations occur in each segment that references objects in other segments or makes dynamic-link references. The `iterated` attribute is displayed for each segment that has iterated data. Iterated data consist of a special code that packs repeated bytes; for example, OS/2 executables produced with the **/EXEPACK** option of **LINK,** have iterated data.

5. Run-time relocations and fixups. See the object-module information in the *Microsoft Operating System/2 Programmer's Reference* for more information.

6. Finally, **EXEHDR** lists all exported entry points. Exports are discussed in Section 3, "About Linking in OS/2," and in Section 7.8, "The EXPORTS Statement."

Section 11

LINK Error Messages

This appendix lists error messages that apply only to the protected-mode version of **LINK**, when used to create protected-mode or Windows files. When you create application under DOS 3.x, you will not receive any of the messages listed below.

Number	**Linker Error Message**
`L1005`	`/PACKCODE : packing limit exceeds 65536 bytes` The value supplied with the **/PACKCODE** option exceeds the limit of 65,536.
`L1030`	`missing internal name` An **IMPORT** statement specified an ordinal in the definitions file without including the internal name of the routine. The name must be given if the import is by ordinal.
`L1031`	`module description redefined` A **DESCRIPTION** in the definitions file was specified more than once, which is not allowed.
`L1032`	`module name redefined` The module name was specified more than once (via a **NAME** or **LIBRARY** statement), which is not allowed.
`L1040`	`too many exported entries` The definitions file exceeded the limit of 3072 exported names.
`L1041`	`resident-name table overflow` The size of the resident-name table exceeds 65,534 bytes. (An entry in the resident-names table is made for each exported routine designated **RESIDENTNAME**, and consists of the name plus three bytes of information. The first entry is the module name.) Reduce the number of exported routines or change some to nonresident.

L1042 `nonresident-name table overflow`

The size of the nonresident-name table exceeds 65,534 bytes. (An entry in the nonresident-names table is made for each exported routine not designated **RESIDENTNAME**, and consists of the name plus three bytes of information. The first entry is the **DESCRIPTION**.) Reduce the number of exported routines or change some to resident.

L1044 `imported-name table overflow`

The size of the imported-names table exceeds 65,534 bytes. (An entry in the imported-names table is made for each new name given in the **IMPORTS** section, including the module names, and consists of the name plus one byte.) Reduce the number of imports.

L1061 `out of memory for /INCREMENTAL`

The linker ran out of memory when trying to process the additional information required for **ILINK** support. If you were linking from within an editor or **MAKE**, try linking directly.

L1062 `too many symbols for /INCREMENTAL`

The program had more symbols than can be stored in the **.SYM** file. Reduce the number of symbols.

L1073 `file-segment limit exceeded`

The number of physical or file segments exceeds the limit of 254 imposed by OS/2 protected mode and by Windows for each application or dynamic-link library. (A file segment is created for each group definition, nonpacked logical segment, and set of packed segments.) Reduce the number of segments or group more of them and make sure that **/PACKCODE** is enabled.

L1074 *name* `: group larger than 64K bytes`

The given group exceeds the limit of 65,536 bytes. Reduce the size of the group, or remove any unneeded segments from the group (look at the map file).

L1075 `entry table larger than 65535 bytes`

The entry table exceeds the limit of 65,535 bytes. (There is a row in this table for each exported routine, and also for each address which is the target of a far relocation and for which one of the following conditions is true: the target segment is designated **IOPL**, or **PROTMODE** is not enabled and the target segment is designated **MOVABLE**.) Declare **PROTMODE** if applicable, or reduce the number of exported routines, or make some segments **FIXED** or **NOIOPL** if possible.

L1082	`stub .EXE file not found` The linker could not open the file given in the **STUB** statement in the definitions file.
L1092	`cannot open module definitions file` The linker could not open the definitions file specified on the command line or in the response file.
L1094	*`file`* `: cannot open file for writing` The linker was unable to open the file with write permission. Check file permissions.
L1095	*`file`* `: out of space on file` The linker ran out of disk space for the specified output file. Create free disk space or delete root directories.
L1100	`stub .EXE file invalid` The file specified in the **STUB** statement is not a valid real-mode executable file.
L1126	`conflicting iopl-parameter-words value` An exported name was specified in the definitions file with an **IOPL**-parameter-words value, and the same name was specified as an export by the Microsoft C **export** pragma with a different parameter-words value.
L2000	`imported starting address` The program starting address as specified in the **END** statement in a **MASM** file is an imported routine. This is not supported in OS/2 or Windows.
L2010	`too many fixups in LIDATA record` The number of far relocations (pointer- or base-type) in an **LIDATA** record, which is typically produced by the **DUP** statement in an .ASM file, exceeds the limit imposed by the linker. The limit is dynamic: a 1024-byte buffer is shared by relocations and by the contents of the **LIDATA** record, and there are eight bytes per relocation. Reduce the number of far relocations in the **DUP** statement.
L2022	*`name`* `(alias` *`internalname`*`) : export undefined` The internal name of the given exported routine is undefined.

L2023 `name (alias internalname) : export imported`

The internal name of the given exported routine conflicts with the internal name of a previously imported routine. The set of imported and exported names can not overlap.

L2026 `entry ordinal number, name name : multiple definitions for same ordinal`

The given exported name with the given ordinal number conflicted with a different exported name previously assigned to the same ordinal. Only one name can be associated with a particular ordinal.

L2027 `name : ordinal too large for export`

The given exported name was assigned an ordinal which exceeded the limit of 3072.

L2028 `automatic data segment plus heap exceed 64K`

The total size of data declared in **DGROUP**, plus the value given in **HEAPSIZE** in the definitions file, plus the stack size given by the **/STACKSIZE** option or **STACKSIZE** definitions file statement, exceeds 64K. Reduce near data allocation, **HEAPSIZE**, or stack.

L2030 `starting address not code (use class 'CODE')`

The program starting address, as specified in the **END** statement of an **.ASM** file, should be in a code segment (code segments are recognized if their class name ends in `'CODE'`). This is an error in OS/2 protected mode; the error message may be disabled by including the statement **REALMODE** in the definitions file.

L4000 `seg disp. included near offset in segment name`

This is the warning generated by the **/WARNFIXUP** option. Refer to documentation on that option.

L4001 `frame-relative fixup, frame ignored near offset in segment name`

A reference is made relative to a segment which is different from the target segment of the reference. For example, if `_foo` is defined in segment `_TEXT`, the instruction `call DGROUP:_foo` will result in this warning. The frame **DGROUP** is ignored, so the linker will treat the call as if it were `call _TEXT:_foo`.

L4002 `frame-relative absolute fixup near` *`offset`* `in segment` *`name`*

A reference is made similar to the type described in `L4001`, but both segments are absolute (defined with **AT**). It is unclear what this means in OS/2 protected mode or Windows; the linker treats the executable file as if the file were to run in real mode only.

L4010 `invalid alignment specification`

The number specified in the **/ALIGNMENT** option must be a power of 2 in the range 2–32,768 (inclusive).

L4011 `PACKCODE value exceeding 65500 unreliable`

The packing limit specified with the **/PACKCODE** option was between 65,500 and 65,536. Code segments with a size in this range are unreliable on some steppings of the 80286 processor.

L4013 `invalid option for new-format executable file ignored`

The use of overlays and the options **/CPARMAXALLOC, /DSALLOCATION, /NOGROUPASSOCIATION,** are not allowed with either OS/2 protected-mode or Windows executables.

L4014 `invalid option for old-format executable file ignored`

The **/ALIGNMENT** option is invalid for real-mode executables.

L4022 *`group1, group2`* `: groups overlap`

The named groups overlap. (Since a group is assigned to a physical segment, groups cannot overlap with either OS/2 protected-mode or Windows executables.) You should reorganize segments and group definitions so the groups do not overlap. Refer to the map file.

L4023 *`name (internal name)`* `: export internal name conflict`

The internal name of the given exported routine conflicted with the internal name of a previous import definition or export definition.

L4024 *`dynlib.import`* `(`*`name`*`) : multiple definitions for export name`

The given name was exported more than once, which is not allowed.

L4025 *`dynlib.import`* `(`*`name`*`) : import internal name conflict`

The internal name of the given imported routine (*import* is either a name or a number) conflicted with the internal name of a previous export or import.

L4026 `*name* : self-imported`

The given imported routine was imported from the module being linked. This is not supported on some systems.

L4027 `*name* : multiple definitions for import internal-name`

The given internal name was imported more than once. Previous import definitions are ignored.

L4028 `*name* : segment already defined`

The given segment was defined more than once in the **SEGMENTS** statement of the definitions file.

L4029 `*name* : DGROUP segment converted to type data`

The given logical segment in the group **DGROUP** was defined as a code segment. (**DGROUP** cannot contain code segments, because the linker always considers **DGROUP** to be a data segment. The name **DGROUP** is predefined as the automatic data segment.) The linker converts the named segment to type "data."

L4030 `*name* : segment attributes changed to conform with automatic data segment`

The given logical segment in the group **DGROUP** was given sharing attributes (**SHARED/NONSHARED**) which differed from the automatic data attributes as declared by the **DATA** *instance* (**SINGLE/MULTIPLE**). The attributes are converted to conform to those of **DGROUP**. Refer to Error `L4029` for more information on **DGROUP**.

L4032 `*name* : code-group size exceeds 65500 bytes`

The given code group has a size between 65,500 and 65,536 bytes, which is unreliable on some steppings of the 80286 processor.

L4036 `no automatic data segment`

The application did not define a group named **DGROUP**. **DGROUP** has special meaning to the linker, which uses it to identify the automatic or default data segment used by the operating system. Most OS/2 protected-mode and Windows applications require **DGROUP**. This warning will not be issued if **DATA NONE** is declared or if the executable is a dynamic-link library.

L4042 `cannot open old version`

The file specified in the **OLD** statement in the definitions file could not be opened.

L4043	`old version not segmented-executable format`
	The file specified in the **OLD** statement in the definitions file was not a valid OS/2 protected-mode or Windows executable.
L4046	`module name different from output file name`
	The name of the executable as specified in the **NAME** or **LIBRARY** statement is different from the output file name. This may cause problems; you should consult the documentation for your operating system.