Microsoft® CodeView™ and Utilities

Software Development Tools

for the Microsoft Operating System/2

Update

Microsoft Corporation

Document No. 510830018-200-001-1087
Part No. 01033

1.1 Introduction

The Software Development Kit (SDK) for the Microsoft® Operating System/2 (MS® OS/2) update includes several changes to the Microsoft CodeView® debugger and the utilities. This document discusses these changes and, where appropriate, refers to the Microsoft CodeView and Utilities manual.

This update document discusses the following major topics:

- Installation
- Debugging
- Introduction to linking in OS/2
- How to use the OS/2 linker
- The **BIND** utility
- The **IMPLIB** utility
- Using module-definition files
- The **MAKE** utility

Important

Some of the information in this update conflicts with material in the *Microsoft Operating System/2 Programmer's Guide.* In all such cases, consider this update to be correct. The information in this update is more recent and therefore supersedes other material, except for on-line information.

1.2 Installation

The Microsoft Operating System/2 SDK includes two versions of the Microsoft CodeView debugger, one for each MS 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.

The editor's executable file is **SDKED.EXE**, but it may be renamed. If you rename it, be sure to include the new name in the tag section of **TOOLS.INI**. For example, if you change the name to **Z.EXE**, then add the following line to **TOOLS.INI**:

[SDKED Z]

1.3 Debugging

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. The **CVP** debugger takes an additional command-line option for use in debugging dynamic-link modules.
3. The **CVP** debugger can debug multiple-thread programs. In order to deal with the multiple-thread capability of OS/2, the **CVP** debugger has a new command that is not present in the **CV** debugger, and some of the commands for tracing and execution in the **CV** debugger work differently in the **CVP** debugger.

Each of these differences is described in the sections that follow. You should also bear in mind the following general limitations when using the **CVP** debugger 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.
- A program being debugged by the CodeView debugger does not have access to its environment, unless it makes no dynamic-link calls other than calls to the application program interface (API).

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

Using the Debugger's View-Output-Screen Command

When you switch display to the output window (by using the \ command), you don't stay there indefinitely as you would with the real-mode CodeView debugger. Instead, you 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 \ 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 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.

Debugging Dynamic-Link Modules

The protected-mode CodeView debugger (**CVP.EXE**) 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** utility and module-definition sections later in this update.

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, the **CVP** debugger has an additional command-line option that 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 the **/L** option from *file*.

■ Example

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

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

Debugging Multiple-Thread Programs

A program running under OS/2 protected mode has one or more threads. As explained in the *Microsoft Operating System/2 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 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, as shown in the figure below:

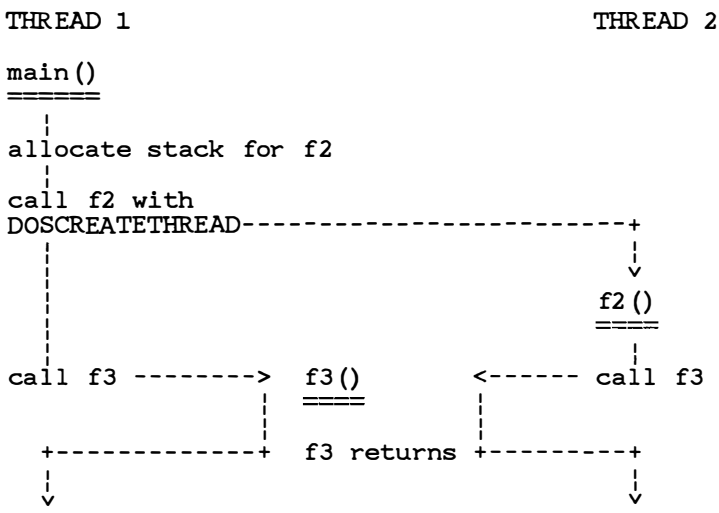

Figure 1.1 Multiple-Thread Example

In the figure, 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 the **CVP** debugger.

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

■ 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 whenever you begin a program. If the program never calls the **DOSCREATETHREAD** function, then thread 1 remains the only thread.

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

- The CodeView prompt displays a different three-digit number.
- The register contents all change.
- The current source line and current instruction both change to reflect the new value of **CS:IP**. If you are running CodeView in window mode, you are likely to see different code in the display window.
- The Calls menu and the Stack Trace command 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 which thread or threads the command applies to. You can specify all threads, or just a particular thread. The *command* determines what activity the debugger carries 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 legal values for *specifier* are listed below, along with their functions:

Specifier	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 functions:

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 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 the **E** command, 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; it executes the current thread in slow motion, but lets all other threads run (except those that are frozen). You see only the current thread executing in the CodeView display.

F The specified thread or threads are frozen.

A frozen thread will not run in the background or in response to the Go command. However, if you use the **E**, **G**, **P**, or **T** variation of the Thread command, then the specified thread is 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 execute concurrently (except for those that are frozen). If you specify a particular thread, then the debugger temporarily freezes all other threads and executes 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 do 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 those that are frozen). You see only the current thread execute in the CodeView 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 variation of the Thread command works in a manner identical to the **P** variation, described above, except that the **T** variation traces through function calls and interrupts, whereas the **P** variation 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 selects that thread as the current thread.

■ Syntax

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

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

In the syntax display above, 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). Also note that the command ~*S is never legal.

■ 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 you can expect the registers and the code displayed all to 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 unfreezes all threads; any threads frozen before are now free to execute whenever the Go command is given. If no threads are frozen, then 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
```

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

Some other CodeView commands are affected by the existence of multiple threads. 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 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 only to the current thread.

■ Thread Termination

After you run the current thread to termination, you can continue to examine registers for the thread until you execute program code. Once you execute program code, the terminated thread disappears. The debugger reports the message `thread terminated normally` when you specify a thread other than the current thread and execute it to termination. The debugger reports the message `program terminated normally` when you execute the current thread to termination.

■ Thread Status

The status display for threads consists of three fields:

ThreadID ThreadState ThreadPriority

The *ThreadID* and *ThreadPriority* fields are integer numbers, and the *ThreadState* field 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.

■ New Error Messages

The following new error messages relate to debugging threads:

`Thread status unavailable with this version of OS/2`

You cannot debug individual threads with this version of OS/2.

`Invalid thread ID`

You gave a thread number that was out of range or corresponds to a thread that no longer exists. Use the ~ command to see what threads exist.

`Thread blocked`

The thread you specified cannot be executed.

`All threads blocked`

All threads are blocked by the operating system or frozen. Check to see if there are any threads that you can unfreeze.

1.4 Introduction to Linking in MS® OS/2

In most respects, linking a program 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 **.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.

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. OS/2 provides two principal ways to supply these definitions:

1. You create a module-definition file 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 a module-definition file 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.

Linking without an Import Library

The figure below illustrates the first method in which each of the two files—the **.EXE** file and the **.DLL** file—has a corresponding module-definition file. (A module-definition file has a **.DEF** default extension.)

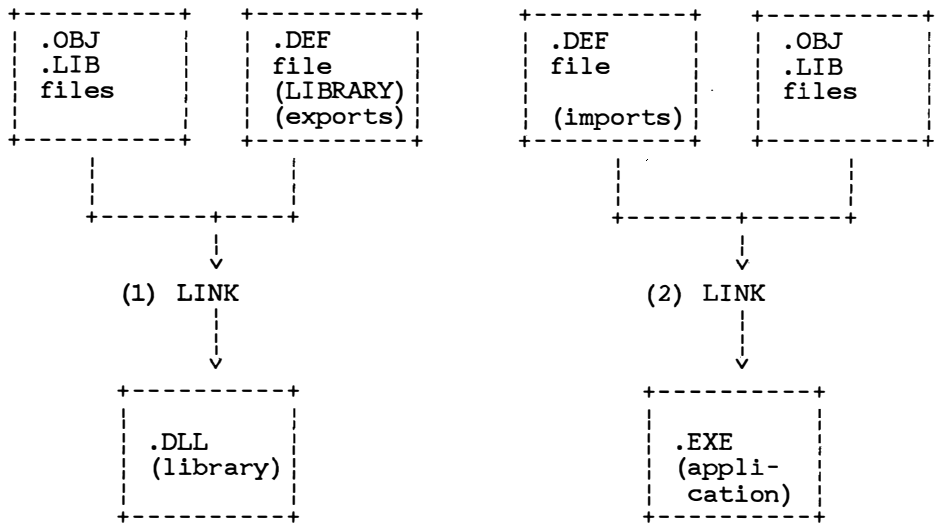

Figure 1.2 Linking without an Import Library

The two major steps are explained below:

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 linker has no way to accept a module-definition file as input. However, the OS/2 linker has an optional field in which you can specify a module-definition file. This field is discussed in Section 1.5, "How to Use the OS/2 Linker."

Linking with an Import Library

The figure below illustrates the second method in which a module-definition file is supplied for the dynamic-link library, and an import library is supplied for the application.

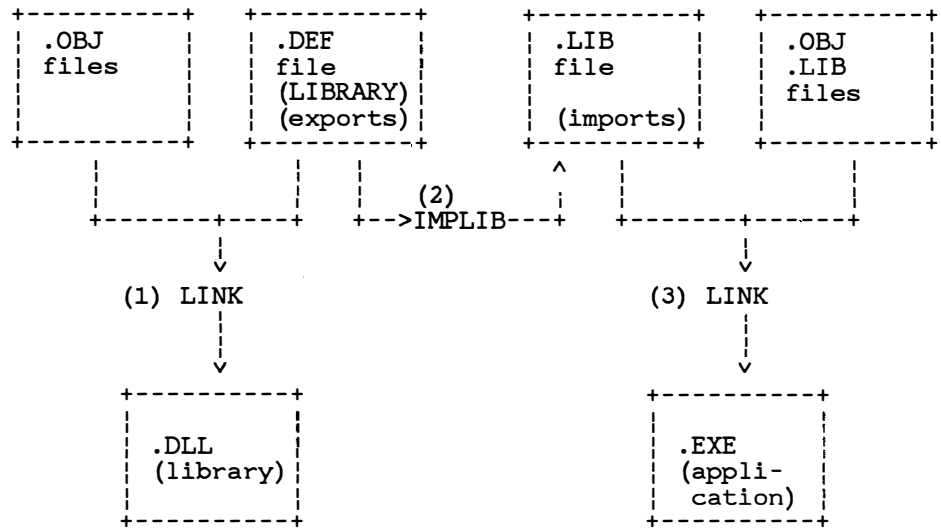

Figure 1.3 Linking with an Import Library

The three major steps are explained below:

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. The **IMPLIB** utility takes as input the same module-definition file used in the first step.

   The **IMPLIB** utility's sole purpose is to generate import definitions in a library-file format. The **IMPLIB** utility 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, the **IMPLIB** utility generates a corresponding import definition.
3. The **.LIB** file generated by the **IMPLIB** utility is used as input to the **LINK** utility, 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 the **IMPLIB** utility 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, but 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. Futhermore, the 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.

Why Use Import Libraries?

At first glance, it may seem easier to create programs without import libraries, since two steps are required rather than three. 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 the **IMPLIB** utility, you specify the **.DEF** file you already created for the dynamic-link library. Operation of the **IMPLIB** utility is simple. If you do not use an import library generated by the **IMPLIB** utility, 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—creation of the **.DLL** file and creation of the import library—are carried out only by the author of the dynamic-link library. The libraries may then be given to an applications developer, who focuses on step 3—linking the application. The application developer's task is simplified if he or she links with the import library, because then the developer does not have to worry about editing his or her 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.

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. You can 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. You can save significant disk space.

   Suppose that 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. You can make libraries and applications more independent.

   Dynamic-link libraries can be updated any number of times without relinking the applications that use it. If you are a third-party supplier of libraries, this fact is particularly convenient. You can ship the updated **.DLL** file to the user, who needs to copy only the new library onto his or her disk. At run time, the user's applications automatically call the updated library functions.

1.5 How to Use the OS/2 Linker

This section describes how to link applications and dynamic-link libraries. It 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 Section 1.4, "Introduction to 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 a dynamic-link library that supports

OS/2 applications. The following rules determine which output the linker produces:

1. If no module-definition file and no import library are given as input, then the linker produces an application for DOS 3.*x*.
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 1.8, "Using Module-Definition Files.") The file **DOSCALLS.LIB** is an import library. Thus, if you link with the **DOSCALLS.LIB** file, you produce either an OS/2 application or a dynamic-link library.

The applications that make dynamic-link calls only to the Family API (a subset of the functions defined in **DOSCALLS.LIB**) can made to run under DOS 3.*x* with the **BIND** utility. The **BIND** utility is discussed in Section 1.6.

■ Syntax

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

LINK *objects*[,[*exe*]][,[*map*]][,[*libraries*]][,*def*][;]

As with the DOS 3.*x* linker, you may specify command-line options after any field but before the comma that terminates the field. Each of the command-line fields is explained below:

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 of more library files that **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 one entry in this field, separate each entry by a space.
def	The file name of a module-definition file. The use of a module-definition file is optional for applications, but required for dynamic-link libraries.

In the list above, 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.

Note

The OS/2 linker supports overlays only when producing a real-mode application. Furthermore, the OS/2 linker does not support nested overlays under any conditions.

The rest of this section discusses linker command-line options.

■ Real-Mode-Only Options

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

Option	Minimum Abbreviation
/CPMAXALLOC	/CP
/DSALLOCATE	/DS
/EXEPACK	/E
/HIGH	/HI
/NOGROUP	/NOG
/OVERLAY	/O

■ Protected-Mode-Only Options

The SDK linker supports two new options that can be used only with protected-mode programs. As mentioned above, most options described in Chapter 12 of the CodeView and Utilities manual can be used with both protected-mode and real-mode programs.

/ALIGNMENT:*size*

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 MS OS/2 application and dynamic-link segments is 512 bytes. Minimum abbreviation is **/A**.

/WARNFIXUP

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 includes the displacement of the segment from the group in determining the final value of the fixup, contrary to what happens with DOS 3.*x* executable files. Minimum abbreviation is **/W**.

■ New Options

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

/NONULLDOSSEG

Directs the linker to put segments in the same order that 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 the **/NONULLDOSSEG** option does not insert these extra bytes.

If the linker is given both the **/DOSSEG** and **/NONULLDOSSEG** options, then the **/NONULLDOSSEG** option always takes precedence. Therefore you can use the **/NONULLDOSSEG** option to override the **DOSSEG** comment record commonly found in run-time libraries. Minimum abbreviation is **/NON**.

/NOEXTDICTIONARY

Prevents the linker from using an extended-dictionary search, which is an internal process the linker uses to save time. Normally, the effect of this option is to slow down the linker. However, you need to use this option whenever a symbol is redefined or when the linker reports that an extended dictionary could not be created. Minimum abbreviation is **/NOE**.

1.6 The BIND Utility

The MS OS/2 **BIND** utility 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 programmer's reference.)

DOS 3.*x* does not support dynamic linking. Therefore, the **BIND** utility must "bind" dynamic-link functions; that is, the utility brings an application program together with emulated versions of the dynamic-link libraries and links them into a single stand-alone file that can run in real mode.

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

1. The **BIND** utility merges the **.EXE** with the appropriate **.LIB** files and loader, as described above.
2. The loader component loads the MS OS/2 **.EXE** file when running MS-DOS 2.*x*/3.*x* and simulates the MS OS/2 startup conditions in an MS-DOS 3.*x* environment.
3. The **API.LIB** library simulates the MS OS/2 Application Program Interface in an MS-DOS 3.*x* environment.

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. This section explains how the **BIND** utility can accommodate these calls.

If your program makes dynamic-link calls to functions outside the API (that is, to functions that are not system calls), use the *linklibs* field described below. The **BIND** utility searches each of the *linklibs* for object code corresponding to the imported functions. In addition, if you are using import definitions with ordinal values 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 1.8, "Using Module-Definition Files.")

Binding Functions in 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 only in protected mode. 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 function of your program to check the operating-system version and ensure that none of these functions are ever called in real mode.

If your program makes either direct or indirect calls to non-Family-API system calls, but you do *not* use the **-n** option, then **BIND** fails to convert your program.

The BIND Command Line

Invoke the **BIND** utility 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.

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. The library **API.LIB** is automatically included in this field.

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 is 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.

BIND Operation

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

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

2. It invokes the linker, giving the **.EXE** 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 that can run in real mode by linking in the loader and the API-simulator routines.
3. It merges the protected-mode and real-mode executable files to produce a single file that can run in either mode.

The Executable File Layout

The MS OS/2 **.EXE** files have two headers. The first header has an MS-DOS 3.*x* format. The second header has the MS OS/2 format. When the **.EXE** file is run on an system, it ignores the MS-DOS 3.*x* header and uses the MS OS/2 format for headers. When run under MS-DOS 3.*x*, the old header is used to load the file. Figure 1.4 shows the arrangement of the merged headers:

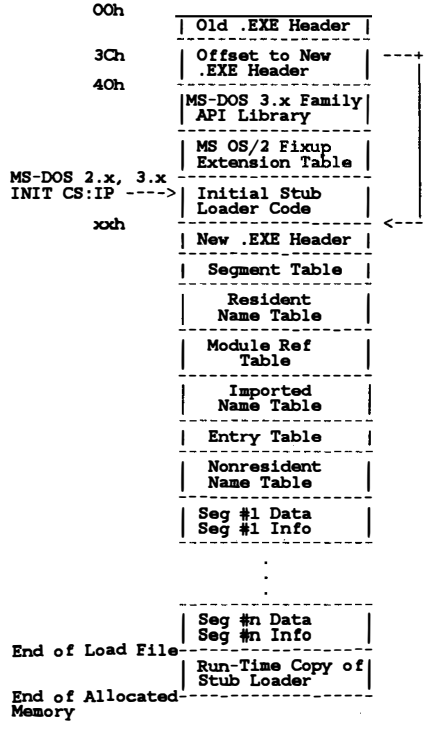

Figure 1.4 Merged Headers in a Bound File

1.7 The IMPLIB Utility

This section summarizes the use of the **IMPLIB** utility. It 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 1.4, "Introduction to Linking in OS/2."

You can create an import library for use by other application developers in resolving external references to your dynamic-link library. The **IMPLIB** command creates an import library, which is a **.LIB** file that can be read by the MS 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 this facility, external references to dynamic-link routines must be declared in an **IMPORTS** statement in the module-definition file for the application being linked.

■ Syntax

IMPLIB *implib-name mod-def-file* [[*mod-def-file...*]]

The *implib-name* 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 MS OS/2 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
```

1.8 Using Module-Definition Files

A module-definition file describes the name, size, format, functions, and segments of an application or library for MS OS/2 or Microsoft Windows. This file is required for Windows applications and libraries, and it is also required for dynamic-link libraries that run under MS 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 below:

Statement	Attribute
NAME	Name and type of application (no library created)
LIBRARY	Name of dynamic-link library (no application created)
DESCRIPTION	One-line description of the module
CODE	Default attributes for code segments
DATA	Default attributes for data segments
SEGMENTS	Attributes for specific segments
STACKSIZE	Local-stack size, in bytes
EXPORTS	Exported functions
IMPORTS	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	Local-heap size, in bytes
PROTMODE	Specifies that the module runs only in OS/2 protected mode
REALMODE	Specifies that the module is for real-mode Windows
OLD	Preserves export ordinal information from a previous version of the library

The following rules govern the use of these statements in a module-definitions 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 all-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 these fields is explained in the sections that follow, each presenting a description of a module-definition statement along with syntax and examples.

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 up to eight characters long. If no

appname is given, the name of the executable file—with the extension removed—becomes the name of the application.

If *apptype* is given, it defines the type of application being linked. This information is kept in the executable-file header and is used by Presentation Manager. The *apptype* field may have one of the following values:

Apptype	Description
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 that are supported in Windows applications.
NOTWINCOMPAT	This 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
```

The LIBRARY Statement

The **LIBRARY** statement identifies the executable file as a dynamic-link library, optionally specifies the name of the library, and optionally specifies 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 up to eight characters long. 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. The *initialization* defaults to **INITGLOBAL** if neither is given.

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 be performed each time a new process gains access to "calendar."

```
LIBRARY calendar INITINSTANCE
```

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'
```

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; 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	**DISCARDABLE, NONDISCARDABLE**

The *load* field determines when a code segment is 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 or not 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 segments have I/O privilege.
NOIOPL	The code segments do not have I/O privilege (the default).

The *conforming* field specifies whether or not a code segment is a 286 "conforming" segment. 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 or not 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 these keywords: **SHARED** or **NONSHARED** (the default).

The *movable* field determines whether or not 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 or not 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, 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 the module's code segments so that they have I/O hardware privilege and are not loaded until accessed:

```
CODE LOADONCALL IOPL
```

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; 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	Value
load	**PRELOAD, LOADONCALL**
readonly	**READONLY, READWRITE**
instance	**NONE, SINGLE, MULTIPLE**
iopl	**IOPL, NOIOPL**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**DISCARDABLE, NONDISCARDABLE**

The *load* field determines when a segment is 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	Set access rights to read only.
READWRITE	Set access rights to both read and write (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 grouping makes up the physical segment that 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	All instances of the module share a single automatic data segment. In this case, the module is said to have "solo" data.
MULTIPLE	The automatic data segment is copied for each instance of the module. In this case, the module is said to have "instance" data (the default).

The *iopl* field determines whether or not data segments have "I/O" privilege; that is, whether or not they have Input/Output privilege and 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 or not all instances of the program can share a **READWRITE** 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 is 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).

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 such as `DATA SINGLE NONSHARED`, all segments in **DGROUP** are shared; all other data segments are nonshared by default. If a segment that is a member of **DGROUP** is defined with a *sharing* attribute conflicting 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
```

The *movable* field determines whether or not 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 or not 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, all segments can be swapped as needed. The field contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

■ Example

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

```
DATA LOADONCALL NONSHARED
```

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

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
segmentname [[**CLASS** '*classname*']] [[*attribute...*]]
.
.
.

■ 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).

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. 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; 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**, **NOIPL**
conforming	**CONFORMING**, **NONCONFORMING**
shared	**SHARED**, **NONSHARED**
movable	**MOVABLE**, **FIXED**
discard	**DISCARDABLE**, **NONDISCARDABLE**

The *load* field determines when a segment is 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	Set access rights to read only.
READWRITE	Set access rights to both read and write (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; in which case the 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 or not all instances of the program can share a **READWRITE** 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 or not 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 or not 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, 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
```

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, and it 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 stack space:

```
STACKSIZE 4096
```

The EXPORTS Statement

The **EXPORTS** statement defines the names and attributes of the functions exported to other modules and 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
entryname[[= *internalname*]] [[@ *ord*]] [[*pwords*]] [[**RESIDENTNAME**]] [[**NODATA**]]
.
.
.

■ 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 *entryname* specification, which defines the function name, is the name of the function 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 position. Use of ordinal positions is faster and may save space.

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 **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 optional keyword **NODATA** has no effect in OS/2, but is provided for use by real-mode Windows. It inhibits prolog editing.

In OS/2, the **EXPORTS** statement is only meaningful for functions within dynamic-link libraries; Windows applications may export functions. For more information, see the *Microsoft Windows Programmer's Reference.*

■ 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 value. 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
```

The IMPORTS Statement

The **IMPORTS** statement defines the names of the functions that will be imported for the application or library. Usually, **LINK** uses the import-library file (created by the **IMPLIB** utility) to resolve external references to functions. However, the **IMPORTS** statement provides an alternative for resolving these references within a module.

■ Syntax

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

■ 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 *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, although 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 has the same name in all three contexts. If either of the modules uses 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
```

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 is invoked whenever the module is executed under DOS 2.*x* or DOS 3.*x*. Typically, the stub displays a message and terminates execution.

■ 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 **STOPIT.EXE** to the beginning of the module:

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

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

The HEAPSIZE Statement

The **HEAPSIZE** statement defines the size of the application's local heap, in bytes.

■ 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. This statement has no effect if the module is a dynamic-link library.

■ Example

```
HEAPSIZE 4000
```

The PROTMODE Statement

The **PROTMODE** statement specifies that the module will run only in protected mode.

■ Syntax

PROTMODE

■ Remarks

If this statement is not included in the module-definition file, it is assumed that the application may be bound. See Section 1.6 on the **BIND** utility statement for more information.

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** statement has no effect on application modules.

1.9 Using the /X Option with the MAKE Utility

In addition to the options listed in Section 14.5, "Specifying MAKE Options," in the CodeView and Utilities manual, the version of the **MAKE** utility supplied with MS OS/2 SDK has an additional option that redirects error output.

■ Syntax

/X *file*

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

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

For example, the **MAKE** utility issues this message 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 that 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**.