C H A P T E R 8

Sharing Data and Procedures Among Modules and Libraries

To use symbols and procedures in more than one module, the assembler must be able to recognize the shared data as global to all the modules where they are used. MASM provides techniques to simplify data-sharing and give a high-level interface to multiple-module programming. With these techniques, you can place shared symbols in include files. This makes the data declarations in the file available to all modules that use the include file.

This chapter explains the two data-sharing methods MASM 6.1 offers. The first method simplifies data sharing between modules with include files. The second does not involve include files. Instead, this method allows modules to share procedures and data items using the **PUBLIC** and **EXTERN** directives.

The last section of this chapter explains how to create program libraries and access their routines.

Selecting Data-Sharing Methods

If data defined in one module is to be used in other modules of a program, you must declare the data public and external. MASM provides several ways to do this:

- Declare a symbol public with the **PUBLIC** directive in the module where it is defined. This makes the symbol available to other modules. You must also place an **EXTERN** statement for that symbol in all other modules that refer to the public symbol. This statement informs the assembler that the symbol is external—that is, defined in another module.
- Declare the data communal with the **COMM** directive. However, communal variables have limitations. You cannot depend on their location in memory because they are allocated by the linker, and they cannot be initialized.

The **EXTERNDEF** directive declares a symbol either public or external, as appropriate. **EXTERNDEF** simplifies the declarations for global (public and external) variables and encourages the use of include files.

The next section provides further details on using include files. For more information on **PUBLIC** and **EXTERN**, see "Using Alternatives to Include Files," page 219.

Sharing Symbols with Include Files

Include files can contain any valid MASM statement, but typically consist of type and symbol declarations. The assembler inserts the contents of the include file into a module at the location of the **INCLUDE** directive. Include files are optional, but can simplify project organization by eliminating the need to insert common declarations into all modules of a program. An alternative to using include files is described in "Using Alternatives to Include Files," page 219.

This section explains how to organize symbol definitions and the declarations that make them global (available to all modules); how to make both variables and procedures public with **EXTERNDEF**, **PROTO**, and **COMM**.; and where to place these directives in the modules and include files.

Organizing Modules

This section summarizes the organization of declarations and definitions in modules and include files and the use of the **INCLUDE** directive.

Include Files

Type declarations that need to be identical in every module should be placed in an include file. This ensures consistency and saves time when you update programs. Include files should contain only symbol declarations and any other declarations that are resolved at assembly time. (For a list of assembly-time operations, see "Generating and Running Executable Programs" in Chapter 1.)

If more than one module accesses the include file, the file cannot contain statements that define and allocate memory for symbols. Otherwise, the assembler would attempt to allocate the same symbol more than once.

Note An include file used in two or more modules should not allocate data variables.

Modules

An **INCLUDE** statement is usually placed before data and code segments in your modules. When the assembler encounters an **INCLUDE** directive, it opens the specified file and assembles all its statements. The assembler then returns to the original module and continues the assembly.

The **INCLUDE** directive takes the form:

INCLUDE *filename*

where *filename* is the full name of the include file. For example, the following declaration inserts the contents of the include file SCREEN.INC in your program:

```
INCLUDE SCREEN.INC
```

The filename in the **INCLUDE** directive must be fully specified; no extensions are assumed. If a full pathname is not given, the assembler first searches the directory of the source file containing the **INCLUDE** directive.

If the include file is not in the source file directory, the assembler searches the paths specified in the assembler's command-line option /I, or in PWB's Include Paths field in the MASM Option dialog box (accessed from the Option menu). The /I option takes this form:

/I *path*

You can include more than one /I option on the command line. The assembler then searches for include files within each specified path in the order given. If none of these directories contains the include file, the assembler finally searches in the paths specified in the INCLUDE environment variable. If the include file still cannot be found, an assembly error occurs. (The /x command-line option tells the assembler to ignore the INCLUDE environment variable when searching for include files.)

An include file may specify another include file. The assembler processes the second include file before returning to the first. Your program can nest include files this way as deeply as the amount of free memory allows.

Include Files or Modules

You can use the **EQU** directive to create named constants that cannot be redefined in your program. (For information about the **EQU** directive, see "Integer Constants and Constant Expressions," page 11.) Placing a constant defined with **EQU** in an include file makes it available to all modules that use that include file.

Placing **TYPEDEF**, **STRUCT**, **UNION**, and **RECORD** definitions in an include file guarantees consistency in type definitions. If required, the variable instances derived from these definitions can be made public among the modules with **EXTERNDEF** declarations (see the next section). Macros, including macros defined with **TEXTEQU**, must be placed in include files to make them visible in other modules.

If you elect to use full segment definitions with, or instead of, simplified definitions, you can force a consistent segment order in all files by defining segments in an include file. This technique is explained in "Controlling the Segment Order,"
page 47.

Declaring Symbols Public and External

It is sometimes useful to make certain procedures and variables (such as status flags) global to all program modules. Global variables are freely accessible within all routines; you do not have to explicitly pass them to the routines that need them. This section describes how to make variables and procedures global using the **EXTERNDEF**, **PROTO**, or **COMM** declarations within include files.

When a procedure is defined in one module and called in another module, it must be declared public in the defining module and external in the calling module(s). MASM offers three ways to declare a procedure public and external:

- Use the **PUBLIC** directive in the defining module and **EXTERN** in all other modules that reference the procedure. The **PUBLIC** and **EXTERN** directives are explained on page 220.
- Declare the procedure with **EXTERNDEF**.
- Prototype the procedure with the **PROTO** directive.

Using EXTERNDEF

MASM treats **EXTERNDEF** as a public declaration in the defining module, and as an external declaration in the referencing module(s). You can use the **EXTERNDEF** statement in your include file to make a variable common to two or more modules. **EXTERNDEF** works with all types of variables, including arrays, structures, unions, and records. It also works with procedures.

As a result, a single include file can contain an **EXTERNDEF** declaration that works in both the defining module and any referencing module. It is ignored in modules that neither define nor reference the variable. Therefore, an include file for a library which is used in multiple .EXE files does not force the definition of a symbol as **EXTERN** does.

The **EXTERNDEF** statement takes this form:

EXTERNDEF ⟦*langtype*⟧ *name*:*qualifiedtype*

The *name* is the variable's identifier. The *qualifiedtype* is explained in detail in "Data Types," page 14.

The optional *langtype* specifier sets the naming conventions for the *name* it precedes. It overrides any language specified in the **.MODEL** directive. The specifier can be **C**, **SYSCALL**, **STDCALL**, **PASCAL**, **FORTRAN**, or

BASIC. For information on selecting the appropriate *langtype* type, see "Naming and Calling Conventions," page 308.

The following diagram shows the statements that declare an array, make it public, and use it in another module.

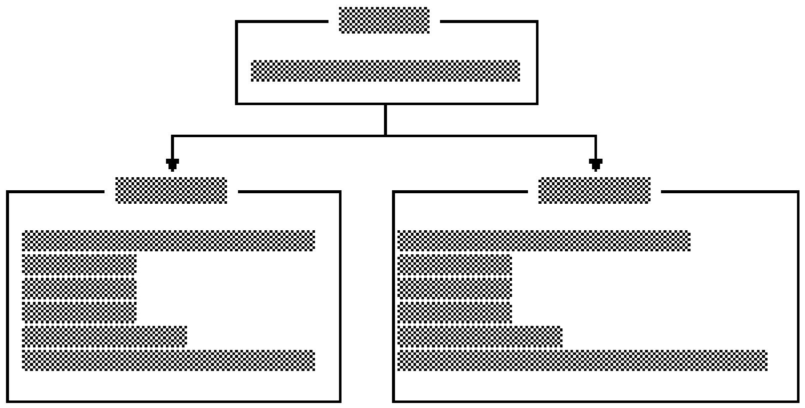

Figure 8.1 Using EXTERNDEF for Variables

The file position of **EXTERNDEF** directives is important. For more information, see "Positioning External Declarations," following.

You can also make procedures visible by using **EXTERNDEF** without **PROTO** inside an include file. This method treats the procedure name as a simple identifier, without the parameter list, so you forgo the assembler's ability to check for the correct parameters during assembly. Use **EXTERNDEF** with procedures in the same way as variables:

```
EXTERNDEF MyProc:FAR            ; Declare far procedure external
```

You can also use **EXTERNDEF** to make a code label global between modules so that one module can reference a label in another module. Give the label global scope with the double colon operator, like this:

```
EXTERNDEF codelabel:NEAR
.
.
.
codelabel::
```

Another module can reference `codelabel` like this:

```
EXTERNDEF codelabel:NEAR
.
.
.
        jmp     codelabel
```

Using PROTO

This section describes how to prototype a procedure with the **PROTO** directive. **PROTO** automatically issues an **EXTERNDEF** for the procedure unless the **PROC** statement declares the procedure **PRIVATE**. Defining a prototype enables type-checking for the procedure arguments.

Follow these steps to create an interface for a procedure defined in one module and called from other modules:

1. Place the **PROTO** declaration in the include file.
2. Define the procedure with **PROC** in one module. The **PROC** directive declares the procedure **PUBLIC** by default.
3. Call the procedure with the **INVOKE** statement (or with **CALL**). Make sure that all calling modules access the include file.

For descriptions, syntax, and examples of **PROTO**, **PROC**, and **INVOKE,** see Chapter 7, "Controlling Program Flow."

The following example illustrates these three steps. In the example, a **PROTO** statement defines the far procedure `CopyFile`, which uses the C parameter-passing and naming conventions, and takes the arguments `filename` and `numberlines`. The diagram following the example shows the file placement for these statements.

This definition goes into the include file:

```
CopyFile PROTO FAR C filename:BYTE, numberlines:WORD
```

The procedure definition for `CopyFile` is:

```
CopyFile PROC FAR C USES cx, filename:BYTE, numberlines:WORD
```

To call the `CopyFile` procedure, you can use this **INVOKE** statement:

```
        INVOKE   CopyFile, NameVar, 200
```

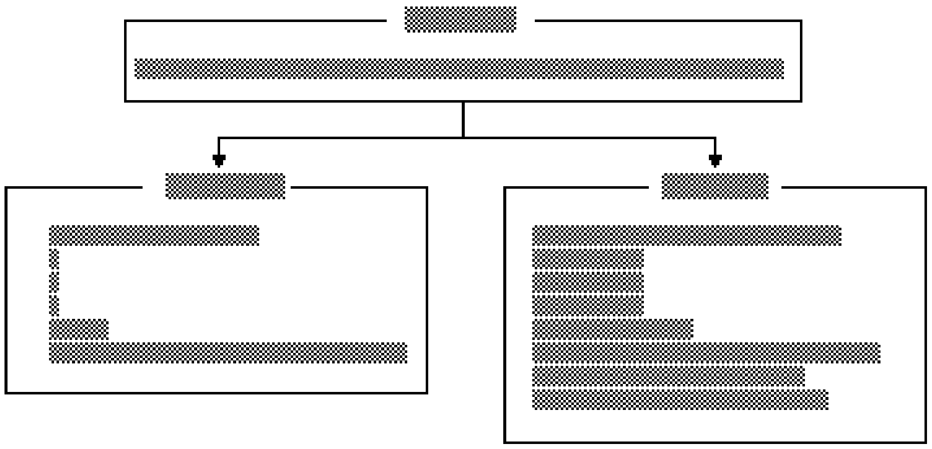

Figure 8.2 Using PROTO and INVOKE

Using COMM

Another way to share variables among modules is to add the **COMM** (communal) declaration to your include file. Since communal variables are allocated by the linker and cannot be initialized, you cannot depend on their location or sequence.

Communal variables are supported by MASM primarily for compatibility with communal variables in Microsoft C. Communal variables are not used in any other Microsoft language, and they are not compatible with C++ and some other languages.

COMM declares a data variable external and instructs the linker to allocate the variable if it has not been explicitly defined in a module. The memory space for communal variables may not be assigned until load time, so using communal variables may reduce the size of your executable file.

The **COMM** declaration has the syntax:

COMM ⟦*langtype*⟧ ⟦**NEAR** | **FAR**⟧ *label*:*type*⟦:*count*⟧

The *label* is the name of the variable. The *langtype* sets the naming conventions for the name it precedes. It overrides any language specified in the **.MODEL** directive.

If **NEAR** or **FAR** is not specified, the variable determines the default from the current memory model (**NEAR** for **TINY**, **SMALL**, **COMPACT**, and **FLAT**; **FAR** for **MEDIUM**, **LARGE**, and **HUGE**). If you do not provide a memory model with the .**MODEL** directive, you must specify a distance when accessing a communal variable, like this:

```
        mov     ax, NEAR PTR CommNear
        mov     bx, FAR PTR CommFar
```

The *type* can be a constant expression, but it is usually a type such as **BYTE**, **WORD**, or **DWORD**, or a structure, union, or record. If you first declare the *type* with **TYPEDEF**, CodeView can provide type information. The *count* is the number of elements. If no *count* is given, one element is assumed.

The following example creates the on far variable `DataBlock`, which is a 1,024-element array of uninitialized signed doublewords:

```
COMM FAR DataBlock:SDWORD:1024
```

Note C variables declared outside functions (except static variables) are communal unless explicitly initialized; they are the same as assembly-language communal variables. If you are writing assembly-language modules for C, you can declare the same communal variables in both C and MASM include files. However, communal variables in C do not have to be declared communal in assembler. The linker will match the **EXTERN**, **PUBLIC**, and **COMM** statements for the variable.

EXTERNDEF (explained in the previous section) is more flexible than **COMM** because you can initialize variables defined with it, and your code can rely on the position and sequence of the defined data.

Positioning External Declarations

Although LINK determines the actual address of an external symbol, the assembler assumes a default segment for the symbol, based on the location of the external directive in the source code. You should therefore position **EXTERN** and
EXTERNDEF directives according to these rules:

- If you know which segment defines an external symbol, put the **EXTERN** statement in that segment.

- If you know the group but not the segment, position the **EXTERN** statement outside any segment and reference the variable with the group name. For example, if `var1` is in DGROUP, reference the variable as

```
mov DGROUP:var1, 10
```

- If you know nothing about the location of an external variable, put the **EXTERN** statement outside any segment. You can use the **SEG** directive to access the external variable like this:

```
mov     ax, SEG var1
mov     es, ax
mov     ax, es:var1
```

- If the symbol is an absolute symbol or a far code label, you can declare it external anywhere in the source code.

Always close any segments opened in include files so that external declarations following an include statement are not incorrectly placed inside a segment. If you want to be certain an external definition lies outside a segment, you can use **@CurSeg**. The **@CurSeg** predefined symbol returns a blank if the definition is not in a segment. For example,

```
        .DATA
        .
        .
        .
@CurSeg ENDS                    ; Close segment
        EXTERNDEF var:WORD
```

For information about predefined symbols such as **@CurSeg**, see “Predefined Symbols,” page 10.

Using Alternatives to Include Files

If your project uses only two modules (or if it is written with a version of MASM prior to 6.0), you may want to continue using **PUBLIC** in the defining module and **EXTERN** in the referencing module, and not create an include file for the project. The **EXTERN** directive can be used in an include file, but the include file containing **EXTERN** cannot be added to the module that contains the corresponding **PUBLIC** directive for that symbol. This section assumes that you are not using include files.

PUBLIC and EXTERN

The **PUBLIC** and **EXTERN** directives are less flexible than **EXTERNDEF** and **PROTO** because they are module-specific: **PUBLIC** must appear in the defining module and **EXTERN** must appear in the calling modules. This section shows how to use **PUBLIC** and **EXTERN**. Information on where to place the external declarations in your file is in "Positioning External Declarations," previous.

The **PUBLIC** directive makes a name visible outside the module in which it is defined. This gives other program modules access to that identifier.

The **EXTERN** directive performs the complementary function. It tells the assembler that a name referenced within a particular module is actually defined and declared public in another module that will be specified at link time.

A **PUBLIC** directive can appear anywhere in a file. Its syntax is:

PUBLIC ⟦*langtype*⟧ *name*⟦**,** ⟦*langtype*⟧ *name*⟧...

The *name* must be the name of an identifier defined within the current source file. Only code labels, data labels, procedures, and numeric equates can be declared public.

If you specify the *langtype* field here, it overrides the language specified by **.MODEL**. The *langtype* field can be **C**, **SYSCALL**, **STDCALL**, **PASCAL**, **FORTRAN**, or **BASIC**. For more information on specifying *langtype* types, see "Declaring Parameters with the PROC Directive," page 184, and "Naming and Calling Conventions," page 308.

The **EXTERN** directive tells the assembler that an identifier is external—defined in some other module that will be supplied at link time. Its syntax is:

EXTERN ⟦*langtype*⟧ *name***:{ABS** | *qualifiedtype*}

"Data Types," page 14, describes *qualifiedtype*. You can use the **ABS** (absolute) keyword only with external numeric constants. **ABS** causes the identifier to be imported as a relocatable unsized constant. This identifier can then be used anywhere a constant can be used. If the identifier is not found in another module at link time, the linker generates an error.

In the following example, the procedure `BuildTable` and the variable `Var` are declared public. The procedure uses the Pascal naming and data-passing conventions:

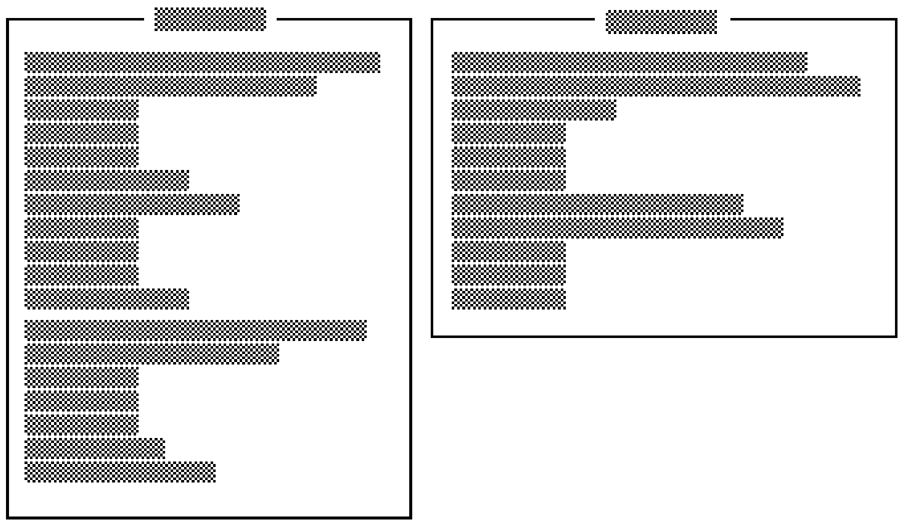

Figure 8.3 Using PUBLIC and EXTERN

Other Alternatives

You can also use the directives discussed earlier (**EXTERNDEF**, **PROTO**, and **COMM**) without the include file. In this case, place the declarations to make a symbol global in the same module where the symbol is defined. You might want to use this technique if you are linking only a few modules that have very little data in common.

Developing Libraries

As you create reusable procedures, you can place them in a library file for convenient access. Although you can put any routine into a library, each library file, recognizable by its .LIB extension, usually contains related routines. For example, you might place string-manipulation functions in one library, matrix calculations in another, and port communications in another. Do not place communal variables (defined with the **COMM** directive) in a library.

A library consists of combined object modules, each created from a single source file. The object module is the smallest independent unit in a library. If you link with one symbol in a module, the linker adds the entire module to your program, but not the entire library.

Associating Libraries with Modules

You can choose either of two methods for associating your libraries with the modules that use them: you can use the **INCLUDELIB** directive inside your source files, or link the modules from the command line.

To associate a specified library with your object code, use **INCLUDELIB**. You can add this directive to the source file to specify the libraries you want linked, rather than specifying them in the LINK command line. The **INCLUDELIB** syntax is:

INCLUDELIB *libraryname*

The *libraryname* can be a file name or a complete path specification. If you do not specify an extension, .LIB is assumed. The *libraryname* is placed in the comment record of the object file. LINK reads this record and links with the specified library file.

For example, the statement `INCLUDELIB GRAPHICS` passes a message from the assembler to the linker telling LINK to use library routines from the file GRAPHICS.LIB. If you place this statement in the source file DRAW.ASM and GRAPHICS.LIB is in the same directory, you can assemble and link the program with the following command:

```
ML DRAW.ASM
```

Without the **INCLUDELIB** directive, you must link the program DRAW.ASM with either of the following commands:

```
ML DRAW.ASM GRAPHICS.LIB
ML DRAW /link GRAPHICS
```

If you want to assemble and link separately, type

```
ML /c DRAW.ASM
LINK DRAW,,,GRAPHICS
```

If you do not specify a complete path in the **INCLUDELIB** statement or at the command line, LINK searches for the library file in the following order:

1. In the current directory.
2. In any directories in the library field of the LINK command line.
3. In any directories specified by the LIB environment variable.

The LIB.EXE utility helps you create, organize, and maintain run-time libraries. Refer to *Environment and Tools* for instructions on LIB.EXE.

Using EXTERN with Library Routines

In some cases, **EXTERN** helps you limit the size of your executable file by specifying in the syntax an alternative name for a procedure. You would use this form of the **EXTERN** directive when declaring a procedure or symbol that may not need to be used.

The syntax looks like this:

EXTERN ⟦*langtype*⟧ *name* ⟦ **(***altname***)** ⟧ **:***qualifiedtype*

The addition of the *altname* to the syntax provides the name of an alternate procedure that the linker uses to resolve the external reference if the procedure given by *name* is not needed. Both *name* and *altname* must have the same *qualifiedtype*.

When the linker encounters an external definition for a procedure that gives an *altname*, the linker finishes processing that module before it links the object module that contains the procedure given by *name*. If the program does not reference any symbols in the *name* file's object from any of the linked modules, the linker uses *altname* to satisfy the external reference. This saves space because the library object module is not brought in.

For example, assume that the contents of STARTUP.ASM include these statements:

```
EXTERN  init(dummy):PROC
        .
        .
        .
dummy   PROC
        .
        .
        .                       ; A procedure definition containing no
        ret                     ;   executable code

dummy   ENDP
        .
        .
        .
        call   init             ; Defined in FLOAT.OBJ
```

In this example, the reference to the routine `init` (defined in FLOAT.OBJ) does not force the module FLOAT.OBJ to be linked into the executable file. If another reference causes FLOAT.OBJ to be linked into the executable file, then `init` will refer to the `init` label in FLOAT.OBJ. If there are no references that force linkage with FLOAT.OBJ, the linker will use the alternate name for `init(dummy)`.