C H A P T E R

4

Units and Related Mysteries

In Chapter 3, you learned how to write standard Pascal programs. What about non-standard programming—more specifically, PC-style programming, with screen control, DOS calls, and graphics? To write such programs, you have to understand units or understand the PC hardware enough to do the work yourself. This chapter explains what a unit is, how you use it, what predefined units are available, how to go about writing your own units, and how to compile them.

What's a Unit, Anyway?

Turbo Pascal gives you access to a large number of predefined constants, data types, variables, procedures, and functions. Some are specific to Turbo Pascal; others are specific to the IBM PC (and compatibles) or to MS-DOS. There are dozens of them, but you seldom use them all in a given program. Because of this, they are split into related groups called *units*. You can then use only the units your program needs.

A *unit* is a collection of constants, data types, variables, procedures, and functions. Each unit is almost like a separate Pascal program: It can have a main body that is called before your program starts and does whatever initialization is necessary. In short, a unit is a library of declarations you can pull into your program that allow your program to be split up and separately compiled.

All the declarations within a unit are usually related to one another. For example, the *Crt* unit contains all the declarations for screen-oriented routines on your PC.

Turbo Pascal provides seven standard units for your use. Five of them—*System, Graph, Dos, Crt,* and *Printer*—provide support for your regular Turbo Pascal programs. The other two—*Turbo3* and *Graph3* are designed to help maintain compatibility with programs and data files created under version 3.0 of Turbo Pascal. All seven are stored in the file TURBO.TPL. Some of these are explained more fully in Chapter 5, but we'll look at each one here and explain its general function.

A Unit's Structure

A unit provides a set of capabilities through procedures and functions—with supporting constants, data types, and variables—but it hides how those capabilities are actually implemented by separating the unit into two sections: the *interface* and the *implementation.* When a program uses a unit, all the unit's declarations become available, as if they had been defined within the program itself.

A unit's structure is not unlike that of a program, but with some significant differences. Here's a unit, for example:

```
unit <identifier>;
interface
uses <list of units>;              { Optional }
  { public declarations }
implementation
  { private declarations }
  { procedures and functions }

begin
  { initialization code }
end.
```

The unit header starts with the reserved word **unit**, followed by the unit's name (an identifier), exactly like a program has a name. The next item in a unit is the keyword **interface**. This signals the start of the interface section of the unit—the section visible to any other units or programs that use this unit.

A unit can use other units by specifying them in a **uses** clause. If present, the **uses** clause appears immediately after the keyword **interface**. Note that the general rule of a **uses** clause still applies: If a unit named in a **uses**

clause employs other units, those units must be named in the **uses** clause, and their names must appear in the list *before* the unit using them.

Interface Section

The interface portion—the "public" part—of a unit starts at the reserved word **interface**, which appears after the unit header and ends when the reserved word **implementation** is encountered. The interface determines what is "visible" to any program (or other unit) using that unit; any program using the unit has access to these "visible" items.

In the unit interface, you can declare constants, data types, variables, procedures, and functions. As with a program, these can be arranged in any order, and sections can repeat themselves (for example, **type** ... **var** ... <*procs*> ... **const** ... **type** ... **const** ... **var**).

The procedures and functions visible to any program using the unit are declared here, but their actual bodies—implementations—are found in the implementation section. If the procedure (or function) is *external*, the keyword **external** should appear in the interface, and a second declaration of the procedure need not occur in the implementation. If the procedure (or function) is an inline directive, the machine code (list of integer constants) will appear in the interface section, and another declaration of the procedure cannot occur in the implementation. **forward** declarations are neither necessary nor allowed. The bodies of all the regular procedures and functions are held in the implementation section after all the procedure and function headers have been listed in the interface section.

Implementation Section

The implementation section—the "private" part—starts at the reserved word **implementation**. Everything declared in the interface portion is visible in the implementation: constants, types, variables, procedures, and functions. Furthermore, the implementation can have additional declarations of its own, although these are not visible to any programs using the unit. The program doesn't know they exist and can't reference or call them. However, these hidden items can be (and usually are) used by the "visible" procedures and functions—those routines whose headers appear in the interface section.

If any procedures have been declared external, one or more {$L *filename*} directive(s) should appear anywhere in the source file. If there is no initialization section, then the {$L *filename*} directive can be anywhere before

the final **end** of the unit. The *$L* directive lets you link in assembly language object modules that resolve the external procedures.

The normal procedures and functions declared in the interface—those that are not inline—must reappear in the implementation. The **procedure/function** header that appears in the implementation should either be identical to that which appears in the interface or should be in the short form. For the short form, type in the keyword (**procedure** or **function**), followed by the routine's name (identifier). The routine will then contain all its local declarations (labels, constants, types, variables, and nested procedures and functions), followed by the main body of the routine itself. Say the following declarations appear in the interface of your unit:

```
procedure ISwap(var V1,V2 : integer);
function IMax(V1,V2 : integer) : integer;
```

The implementation could look like this:

```
procedure ISwap;
var
  Temp : integer;
begin
   Temp := V1; V1 := V2; V2 := Temp
end;  { of proc Swap }

function IMax(V1,V2 : integer) : integer;
begin
  if V1 > V2
    then IMax := V1
    else IMax := V2
end;  { of func Max }
```

Routines local to the implementation (that is, not declared in the interface section) must have their complete **procedure/function** header intact.

Initialization Section

The entire implementation portion of the unit is normally bracketed within the reserved words **implementation** and **end**. However, if you put the reserved word **begin** before **end**, with statements between the two, the resulting compound statement—looking very much like the main body of a program—becomes the *initialization* section of the unit.

The initialization section is where you initialize any data structures (variables) that the unit uses or makes available (through the interface) to the program using it. You can use it to open files for the program to use later. For example, the standard unit *Printer* uses its initialization section to

make all the calls to open (for output) the text file *Lst*, which you can then use in your program's *Write* and *Writeln* statements.

When a program using that unit is executed, the unit's initialization section is called before the program's main body is run. If the program uses more than one unit, each unit's initialization section is called (in the order specified in the program's **uses** statement) before the program's main body is executed.

How Are Units Used?

The units your program uses have already been compiled, stored as machine code not Pascal source code; they are not Include files. Even the interface section is stored in the special binary symbol table format that Turbo Pascal uses. Furthermore, certain standard units are stored in a special file (TURBO.TPL) and are automatically loaded into memory along with Turbo Pascal itself.

As a result, using a unit or several units adds very little time (typically less than a second) to the length of your program's compilation. If the units are being loaded in from a separate disk file, a few additional seconds may be required because of the time it takes to read from the disk.

As stated earlier, to use a specific unit or collection of units, you must place a **uses** clause at the start of your program, followed by a list of the unit names you want to use, separated by commas:

```
program MyProg;
uses thisUnit,thatUnit,theOtherUnit;
```

When the compiler sees this **uses** clause, it adds the interface information in each unit to the symbol table and links the machine code that is the implementation to the program itself.

The units are added to the symbol table in the order given; this ordering can be important when one unit uses another unit. For example, if *thisUnit* used *thatUnit*, the **uses** clause would be

```
uses thatUnit,thisUnit,theOtherUnit;
```

or

```
uses thatUnit,theOtherUnit,thisUnit;
```

In short, a unit must be listed after any units it uses.

If you don't put a **uses** clause in your program, Turbo Pascal links in the *System* standard unit anyway. This unit provides some of the standard Pascal routines as well as a number of Turbo Pascal-specific routines.

Referencing Unit Declarations

Once you include a unit in your program, all the constants, data types, variables, procedures, and functions declared in that unit's interface become available to you. For example, suppose the following unit existed:

```
unit MyStuff;
interface
  const
    MyValue  = 915;
  type
    MyStars  = (Deneb,Antares,Betelgeuse);
  var
    MyWord   : string[20];

  procedure SetMyWord(Star : MyStars);
  function  TheAnswer : integer;
```

What you see here is the unit's interface, the portion that is visible to (and used by) your program. Given this, you might write the following program:

```
program TestStuff;
uses MyStuff;
var
  I     : integer;
  AStar : MyStars;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I)
end.
```

Now that you have included the statement **uses** *MyStuff* in your program, you can refer to all the identifiers declared in the interface section in the interface of *MyStuff* (*MyWord*, *MyValue*, and so on). However, consider the following situation:

```
program TestStuff;
uses MyStuff;
const
  MyValue   = 22;
```

```
var
  I      : integer;
  AStar  : MyStars;

  function TheAnswer : integer;
  begin
    TheAnswer := -1
  end;

begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I)
end.
```

This program redefines some of the identifiers declared in *MyStuff*. It will compile and run, but will use its own definitions for *MyValue* and *TheAnswer*, since those were declared more recently than the ones in *MyStuff*.

You're probably wondering whether there's some way in this situation to still refer to the identifiers in *MyStuff*? Yes, preface each one with the identifier *MyStuff* and a period (.). For example, here's yet another version of the earlier program:

```
program TestStuff;
uses MyStuff;
const
  MyValue = 22;
var
  I     : integer;
  AStar : MyStars;

  function TheAnswer : integer;
  begin
    TheAnswer := -1;
  end;

begin
  Writeln(MyStuff.MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := MyStuff.TheAnswer
  Writeln(I)
end.
```

This program will give you the same answers as the first one, even though you've redefined *MyValue* and *TheAnswer*. Indeed, it would have been perfectly legal (although rather wordy) to write the first program as follows:

```
program TestStuff;
uses MyStuff;
var
  I     : integer;
  AStar : MyStuff.MyStars;

begin
  Writeln(MyStuff.MyValue);
  AStar := MyStuff.Deneb;
  MyStuff.SetMyWord(AStar);
  Writeln(MyStuff.MyWord);
  I := MyStuff.TheAnswer;
  Writeln(I)
end.
```

Note that you can preface any identifier—constant, data type, variable, or subprogram—with the unit name.

TURBO.TPL

The file TURBO.TPL contains the standard units: *System*, *Crt*, *Dos*, *Printer*, *Graph*, *Turbo3*, and *Graph3*. These are the units loaded into memory with Turbo Pascal; they're always readily available to you. You will normally keep the file TURBO.TPL in the same directory as TURBO.EXE (or TPC.EXE). However, you can keep it somewhere else, as long as that "somewhere else" is defined as the Turbo directory. That's done using TINST.EXE to install the Turbo directory directly in the TURBO.EXE file.

System **Units used: none**

System contains all the standard and built-in procedures and functions of Turbo Pascal. Every Turbo Pascal routine that is *not* part of standard Pascal and that is *not* in one of the other units is in *System*. This unit is always linked into every program.

Dos

Units used: none

Dos defines numerous Pascal procedures and functions that are equivalent to the most commonly used DOS calls, such as *GetTime, SetTime, DiskSize,* and so on. It also defines two low-level routines, *MsDos* and *Intr,* which allow you to directly invoke any MS-DOS call or system interrupt. *Registers* is the data type for the parameter to *MsDos* and *Intr.* Some other constants and data types are also defined.

Crt

Units used: none

Crt provides a set of PC-specific declarations for input and output: constants, variables, and routines. You can use these to manipulate your text screen (do windowing, direct cursor addressing, text color and background). You can also do "raw" input from the keyboard and control the PC's sound chip. This unit provides a lot of routines that were standard in version 3.0.

Printer

Units used: *Crt*

Printer declares the text-file variable *Lst* and connects it to a device driver that (you guessed it) allows you to send standard Pascal output to the printer using *Write* and *Writeln.* For example, once you include *Printer* in your program, you could do the following:

```
Write(Lst,'The sum of ',A:4,' and ',B:4,' is ');
C := A + B;
Writeln(Lst,C:8);
```

Graph

Units used: *Crt*

Graph supplies a set of fast, powerful graphics routines that allow you to make full use of the graphics capabilities of your PC. It implements the device-independent Borland graphics handler, allowing support of CGA, EGA, Hercules, AT &T 400, MCGA, 3270 PC, and VGA graphics.

Graph3 **Units used:** *Crt*

Graph3 supports the full set of graphics routines—basic, advanced, and turtlegraphics—from version 3.0. They are identical in name, parameters, and function to those in version 3.0.

Turbo3 **Units used:** *Crt*

This unit contains two variables and several procedures that are no longer supported by Turbo Pascal. These include the predefined file variable *Kbd*, the Boolean variable *CBreak*, and the original integer versions of *MemAvail* and *MaxAvail* (which return paragraphs free instead of bytes free, as do the current versions).

Writing Your Own Units

Say you've written a unit called *IntLib*, stored it in a file called INTLIB.PAS, and compiled it to disk; the resulting code file will be called INTLIB.TPU. To use it in your program, you must include a **uses** statement to tell the compiler you're using that unit. Your program might look like this:

```
program MyProg;
uses IntLib;
```

Note that Turbo Pascal expects the unit code file to have the same name (up to eight characters) of the unit itself. If your unit name is *MyUtilities*, then Turbo is going to look for a file called MYUTILIT.PAS. You can override that assumption with the $*U* compiler directive. This directive is passed the name of the .PAS file and must appear just before the unit's name in the **uses** statement. For example, if your program uses *Dos*, *Crt*, and *MyUtilities*, and the last one is stored in a file called UTIL.PAS, then you would write

```
uses Dos, Crt, {$U UTIL.PAS} MyUtilities;
```

Compiling a Unit

You compile a unit exactly like you'd compile a program: Write it using the editor and select the Compile/Compile command (or press *Alt-C*). However, instead of creating an .EXE file, Turbo Pascal will create a .TPU

(Turbo Pascal Unit) file. You can then leave this file as is or merge it into TURBO.TPL using TPUMOVER.EXE (see Chapter 7).

In any case, you probably want to move your .TPU files (along with their source) to the unit directory you specified with the O/D/Unit directories command. That way, you can reference those files without having to give a {$*U*} directive (The Unit directories command lets you give multiple directories for the compiler to search for in unit files.)

You can only have one unit in a given source file; compilation stops when the final **end.** statement is encountered.

An Example

Okay, now let's write a small unit. We'll call it *IntLib* and put in two simple integer routines—a procedure and a function:

```
unit IntLib;
interface
  procedure ISwap(var I,J : integer);
  function  IMax(I,J : integer) : integer;
implementation

procedure ISwap;
var
  Temp  : integer;

begin
  Temp := I; I := J; J := Temp
end;  { of proc ISwap }

function IMax;
begin
  if I > J
    then IMax := I
    else IMax := J
end;  { of func IMax }

end.  { of unit IntLib }
```

Type this in, save it as the file INTLIB.PAS, then compile it to disk. The resulting unit code file is INTLIB.TPU. Move it to your unit directory (whatever that might happen to be).

This next program uses the unit *IntLib*:

```
program IntTest;
uses IntLib;
var
```

```
  A,B  : integer;
begin
  Write('Enter two integer values:  ');
  Readln(A,B);
  ISwap(A,B);
  Writeln('A = ',A,' B = ',B);
  Writeln('The max is ',IMax(A,B));
end.  { of program IntTest }
```

Congratulations! You've just created your first unit!

Units and Large Programs

Up until now, you've probably thought of units only as libraries—collections of useful routines to be shared by several programs. Another function of a unit, however, is to break up a large program into modules.

Two aspects of Turbo Pascal make this modular functionality of units work: (1) its tremendous speed in compiling and linking and (2) its ability to manage several code files simultaneously, such as a program and several units.

Typically, a large program is divided into units that group procedures by their function. For instance, an editor application could be divided into initialization, printing, reading and writing files, formatting, and so on. Also, there could be a "global" unit—one used by all other units, as well as the main program—that defines global constants, data types, variables, procedures, and functions.

The skeleton of a large program might look like this:

```
program Editor;
uses
  Dos,Crt,Printer                          { Standard units from TURBO.TPL }
  EditGlobals,                                      { User-written units }
  EditInit,
  EditPrint,
  EditRead,EditWrite,
  EditFormat;

  { program's declarations, procedures, and functions }

begin  { main program }
end.  { of program Editor }
```

Note that the units in this program could either be in TURBO.TPL or in their own individual .TPU files. If the latter is true, then Turbo Pascal will manage your project for you. This means when you recompile the program Editor, Turbo Pascal will check the last update for each of the .TPU files and recompile them if necessary.

Another reason to use units in large programs has to do with code segment limitations. The 8086 (and related) processors limit the size of a given chunk, or segment, of code to 64K. This means that the main program and any given segment cannot exceed a 64K size. Turbo Pascal handles this by making each unit a separate code segment. Your upper limit is the amount of memory the machine and operating system can support—640K on most PCs. Without units, you're limited to 64K of code for your program. (See Chapter 6, "Project Management," for more information about how to deal with large programs.)

TPUMOVER

You don't have to use a {$*U <filename>*} directive when using the standard runtime units (*System*, *Dos*, and so on). That's because all those units have been moved into the Turbo Pascal unit file (TURBO.TPL). When you compile, those units are always ready to be used when you want them.

Suppose you want to add a well-designed and thoroughly debugged unit to the standard units so that it's automatically loaded into memory when you run the compiler. Is there any way to move it into the Turbo Pascal standard unit library file? Yes, by using the TPUMOVER.EXE utility.

You can also use TPUMOVER to remove units from the Turbo Pascal standard unit library file, reducing its size and the amount of memory it takes up when loaded. (More details on using TPUMOVER can be found in Chapter 7.)

As you've seen, it's really quite simple to write your own units. A well-designed, well-implemented unit simplifies program development; you solve the problems only once, not for each new program. Best of all, a unit provides a clean, simple mechanism for writing very large programs.