C H A P T E R

7

Interfacing Turbo Assembler with Turbo C

While many programmers can—and do—develop entire programs in assembly language, many others prefer to do the bulk of their programming in a high-level language, dipping into assembly language only when low-level control or very high-performance code is required. Still others prefer to program primarily in assembler, taking occasional advantage of high-level language libraries and constructs.

Turbo C lends itself particularly well to supporting mixed C and assembler code on an as-needed basis, providing not one but two mechanisms for integrating assembler and C code. The inline assembly feature of Turbo C provides a quick and simple way to put assembler code directly into a C function. For those who prefer to do their assembler programming in separate modules written entirely in assembly language, Turbo Assembler modules can be assembled separately and linked to Turbo C code.

First, we'll cover the use of inline assembly in Turbo C. Next, we'll discuss the details of linking separately assembled Turbo Assembler modules to Turbo C, and explore the process of calling Turbo Assembler functions from Turbo C code. Finally, we'll cover calling Turbo C functions from Turbo Assembler code. (**Note:** When we refer to Turbo C, we mean versions 1.5 and greater.) Let's begin.

Using inline assembly in Turbo C

If you were to think of an ideal way to use assembler to fine-tune a C program, you would probably ask for the ability to insert assembler instructions at just those critical places in C code where the speed and low-level control of assembler would result in a dramatic improvement in performance. While you're at it, you might as well wish away the traditional complexities of interfacing assembler with C. Better still, you'd like to be able to do all this without changing any other C code one bit, so that already-working C code won't have to be altered.

The high-performance code in Turbo C's libraries is written in inline assembly.

Turbo C fulfills every item on your wish list with inline assembly. Inline assembly is nothing less than the ability to place virtually any assembler code anywhere in your C programs, with full access to C constants, variables, and even functions. In truth, inline assembly is good for more than just fine-tuning, since it's very nearly as powerful as programming strictly in assembler. Inline assembly lets you use just as much or as little assembler in your C programs as you'd like, without having to worry about the details of mixing the two.

Consider the following C code, which is an example of inline assembly:

```
. . .
i = 0;                                      /* set i to 0 (in C) */
asm  dec  WORD PTR i;                /* decrement i (in assembler) */
i++;                                       /* increment i (in C) */
. . .
```

The first and last lines look normal enough, but what is that middle line? As you've probably guessed, the line starting with **asm** is *inline assembly code*. If you were to use a debugger to look at the executable code this C source compiles to, you would find

```
. . .
mov  WORD PTR [bp-02],0000
dec  WORD PTR [bp-02]
inc  WORD PTR [bp-02]
. . .
```

with the inline assembly **DEC** instruction nestled between the compiled code for

```
i = 0;
```

and

```
i++;
```

There are a few limitations on what inline assembler code is allowed to do; see the section "Limitations of inline assembly" on page 274.

Basically, each time the Turbo C compiler encounters the **asm** keyword that indicates inline assembly, it drops the associated assembler line directly into the compiled code with only one change: References to C variables are transformed into the appropriate assembler equivalent, just as the reference to *i* in the preceding example was changed to `WORD PTR [BP-02]`. In short, the **asm** keyword lets you insert virtually any assembler code anywhere in your C code.

The ability to drop assembler code directly into the code Turbo C generates might sound a bit dangerous, and, in truth, inline assembly does have its risks. While Turbo C takes care to compile its code so as to avoid many potentially hazardous interactions with inline assembly, there's no doubt that ill-behaved inline assembly code can cause serious bugs.

On the other hand, any poorly written assembler code, whether it's inline or in a separate module, has the potential to run amuck; that's the price to be paid for the speed and low-level control of assembly language. Besides, bugs are far less common in inline assembly code than in pure assembler code, since Turbo C attends to many programming details, such as entering and exiting functions, passing parameters, and allocating variables. All in all, the ability to easily fine-tune and turbo-charge portions of your C code with inline assembly is well worth the trouble of having to iron out the occasional assembler bug.

Here are some important notes about inline assembly:

1. You must invoke TCC.EXE, the command-line version of Turbo C, in order to use inline assembly. TC.EXE, the user-interface version of Turbo C, does not support inline assembly.
2. It's very possible that the version of TLINK that came with your copy of Turbo Assembler is not the same version that came with your copy of Turbo C. Since important enhancements were made to TLINK in order to support Turbo Assembler, and since further enhancements will no doubt be made, it is important that you link Turbo C modules containing inline assembly with the most recent version of TLINK that you have. The safest way to accomplish this is to make sure that there's only one TLINK.EXE file on the disk you use to run the linker; that TLINK.EXE file should have the latest version number of all the TLINK.EXE files you've received with other Borland products.

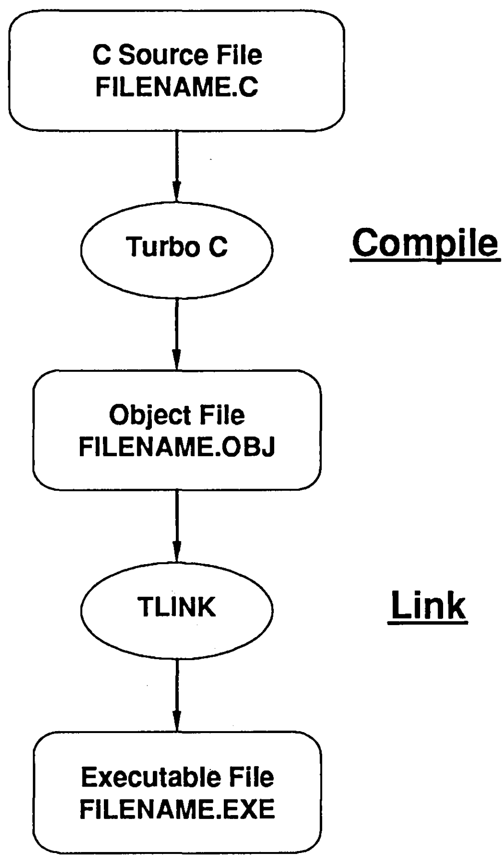
How inline assembly works

Normally, Turbo C compiles each file of C source code directly to an object file, then invokes TLINK to tie the object files together into an executable program. Figure 7.1 shows such a compile-and-link cycle. To start this cycle, you enter the command line

```
tcc filename
```

which instructs Turbo C to first compile FILENAME.C to FILENAME.OBJ and then invoke TLINK to link FILENAME.OBJ into FILENAME.EXE.

Figure 7.1
Turbo C compile and link cycle



When inline assembly is used, however, Turbo C automatically adds one extra step to the compile-and-link sequence.

Turbo C handles each module containing inline assembly code by first compiling the whole module to an assembly language source file, then invoking Turbo Assembler to assemble the resulting assembler code to an object file, and finally invoking TLINK to link the object files together. Figure 7.2 illustrates this process, showing how Turbo C produces an executable file from a C source file containing inline assembly code. You start this cycle with the command line

```
tcc -B filename
```

which instructs Turbo C to first compile FILENAME.ASM, then invoke Turbo Assembler to assemble FILENAME.ASM to FILENAME.OBJ, and finally invoke TLINK to link FILENAME.OBJ into FILENAME.EXE.

Inline assembly code is simply passed along by Turbo C to the assembly language file. The beauty of this system is that Turbo C need not understand anything about assembling the inline code; instead, Turbo C compiles C code to the same level—assembler code—as the inline assembly code and lets Turbo Assembler do the assembling.

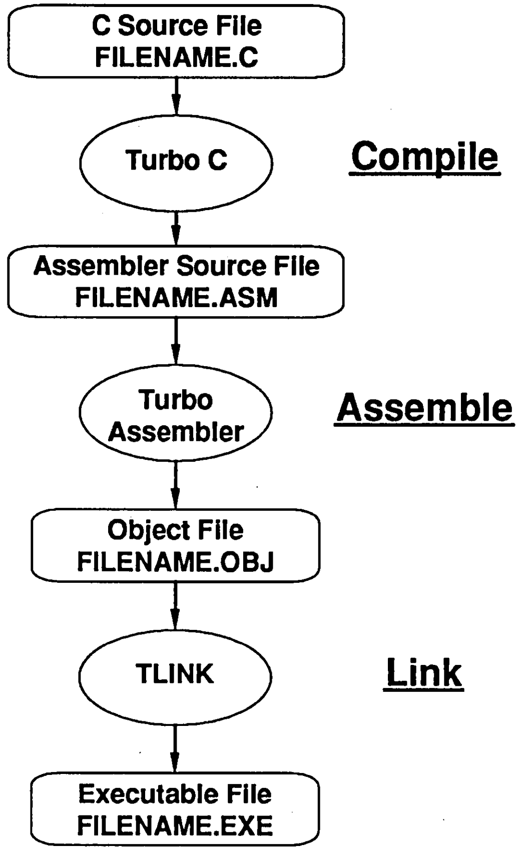
To see exactly how Turbo C handles inline assembly, enter the following program under the name PLUSONE.C (or load it from the example disk):

```
#include <stdio.h>

int  main(void)

{
   int  TestValue;

   scanf("%d",&TestValue);          /* get the value to increment */
   asm  inc  WORD PTR TestValue;    /* increment it (in assembler) */
   printf("%d",TestValue);          /* print the incremented value */
}
```

and compile it with the command line

```
tcc -S plusone
```

Figure 7.2
Turbo C compile, assembly, and link cycle



The **–S** option instructs Turbo C to compile to assembler code and then stop, so the file PLUSONE.ASM should now be on your disk. In PLUSONE.ASM you should find

This code should give you a strong appreciation for all the work Turbo C saves you by supporting inline assembly.

```
        ifndef  ??version
?debug  macro
        ENDM
        ENDIF
        name    Plusone
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
DGROUP  GROUP   _DATA,_BSS
        ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT   ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_d@     LABEL   BYTE
```

```
_d@w    LABEL   WORD
_DATA   ENDS
_BSS    SEGMENT WORD PUBLIC 'BSS'
_b@     LABEL   BYTE
_b@w    LABEL   WORD
        ?debug C E90156E11009706C75736F6E652E63
        ?debug C E90009B9100F696E636C7564655C737464696F2E68
        ?debug C E90009B91010696E636C7564655C7374646172672E68
_BSS    ENDS
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
;       ?debug  L 3
_main   PROC    NEAR
        push    bp
        mov     bp,sp
        dec     sp
        dec     sp
;       ?debug  L 8
        lea     ax,WORD PTR [bp-2]
        push    ax
        mov     ax,OFFSET DGROUP:_s@
        push    ax
        call    NEAR PTR _scanf
        pop     cx
        pop     cx
;       ?debug  L 9
        inc     WORD PTR [bp-2]
;       ?debug  L 10
        push    WORD PTR [bp-2]
        mov     ax,OFFSET DGROUP:_s@+3
        push    ax
        call    NEAR PTR _printf
        pop     cx
        pop     cx
@1:
;       ?debug  L 12
        mov     sp,bp
        pop     bp
        ret
_main   ENDP
_TEXT   ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_s@     LABEL   BYTE
        DB      37
        DB      100
        DB      0
        DB      37
        DB      100
        DB      0
_DATA   ENDS
```

*Here's the assembler code for the **scanf** call, followed by the inline assembler instruction to increment TestValue, followed by the assembler code for the **printf** code.*

Turbo C automatically translates the C variable TestValue to the equivalent assembler addressing of that variable, (BP-2).

```
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
        EXTRN   _printf:NEAR
        EXTRN   _scanf:NEAR
_TEXT   ENDS
        PUBLIC  _main
        END
```

Turbo C compiled the **scanf** call to assembly language, dropped the inline assembly code directly into the assembler output file, and then compiled the **printf** call to assembler. The resulting file is a valid assembler source file, ready to be assembled with Turbo Assembler.

Had you not used the **–S** option, Turbo C would have proceeded to invoke Turbo Assembler to assemble PLUSONE.ASM and would then have invoked TLINK to link the resultant object file, PLUSONE.OBJ, into the executable file PLUSONE.EXE. This is the normal mode of operation of Turbo C with inline assembler; we used **–S** for explanatory purposes only, so that we could examine the intermediate assembly language step Turbo C uses when supporting inline assembly. The **–S** option is not particularly useful when compiling code to be linked into executable programs, but provides a handy means by which to examine both the instructions surrounding your inline assembly code and the code generated by Turbo C in general. If you're ever uncertain about exactly what code you're generating with inline assembly, just examine the .ASM file produced with the **–S** option.

How Turbo C knows to use inline assembly mode

Normally, Turbo C compiles C code directly to object code. There are several ways to tell Turbo C to support inline assembly by compiling to assembly language and then invoking Turbo Assembler.

The **–B** command-line option instructs Turbo C to generate object files by way of compiling to assembler code, then invoking Turbo Assembler to assemble that code.

The **–S** command-line option instructs Turbo C to compile to assembler code, and then stop. The .ASM file generated by Turbo C when the **–S** option is specified can then be separately assembled and linked to other C and assembler modules. Except when debugging or simply exploring, there's generally no reason to use **–S** in preference to **–B**.

The #**pragma** directive

```
#pragma inline
```

has the same effect as the **–B** command-line option, instructing Turbo C to compile to assembly and then invoke Turbo Assembler to assemble the result. When Turbo C encounters **#pragma inline**, compilation restarts in assembler output mode. Consequently, it's best to place the **#pragma inline** directive as close to the start of the C source code as possible, since any C source code preceding **#pragma inline** will be compiled twice, once in normal C-to-object mode and again in C-to-assembler mode. While this doesn't hurt anything, it does waste time.

Finally, if Turbo C encounters inline assembly code in the absence of **–B**, **–S**, and **#pragma inline**, the compiler issues a warning like

```
Warning test.c 6: Restarting compile using assembly in function main
```

and then restarts compilation in assembler-output mode, just as if a **#pragma inline** directive had been encountered at that point. Make it a point to avoid this warning by using the **–B** option or **#pragma inline**, since restarting compilation on encountering inline assembly makes for relatively slow compiles.

Invoking Turbo Assembler for inline assembly

In order for Turbo C to be able to invoke Turbo Assembler, Turbo C must first be able to *find* Turbo Assembler. Exactly how this happens varies with different versions of Turbo C.

Versions of Turbo C later than 1.5 expect to find Turbo Assembler under the file name TASM.EXE in either the current directory or one of the directories pointed to by the DOS PATH environment variable. Basically, Turbo C can invoke Turbo Assembler under the same circumstances in which you could type the command

```
TASM
```

and run Turbo Assembler from the command-line prompt. So, if you have Turbo Assembler in the current directory or anywhere in your command search path, Turbo C will automatically find it and run it to perform inline assembly.

See the README file on the distribution disk for information about how to patch those versions of TCC.

Versions 1.0 and 1.5 of Turbo C behave a little differently. Since these versions of Turbo C were written before Turbo Assembler existed, they invoke MASM, the Microsoft Macro Assembler, to perform inline assembly. Consequently, these versions of Turbo C search the current directory and the command search path for the file MASM.EXE, rather than the file TASM.EXE, and so do not automatically use Turbo Assembler.

Where Turbo C assembles inline assembly

Inline assembly code can end up in either Turbo C's code segment or Turbo C's data segment. Inline assembly code located within a function is assembled into Turbo C's code segment, while inline assembly code located outside a function is assembled into Turbo C's data segment.

For example, the C code

```
/* Table of square values */

asm  SquareLookUpTable  label  word;
asm  dw  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100;

/* Function to look up the square of a value between 0 and 10 */

int LookUpSquare(int Value)
{
   asm  mov  bx,Value;           /* get the value to square */
   asm  shl  bx,1;              /* multiply it by 2 to look up in
                                   a table of word-sized elements */
   asm  mov  ax,[SquareLookUpTable+bx];      /* look up the square */
   return(_AX);                               /* return the result */
}
```

puts the data for *SquareLookUpTable* in Turbo C's data segment and the inline assembly code inside *LookUpSquare* in Turbo C's code segment. The data could equally well be placed in the code segment; consider the following version of *LookUpSquare,* where *SquareLookUpTable* is in Turbo C's code segment:

```
/* Function to look up the square of a value between 0 and 10 */
int LookUpSquare(int Value)
{
   asm  jmp  SkipAroundData            /* jump past the data table */

   /* Table of square values */
   asm  SquareLookUpTable  label  word;
   asm  dw  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100;

SkipAroundData:
   asm  mov  bx,Value;                     /* get the value to square */
   asm  shl  bx,1;                     /* multiply it by 2 to look up
                                  in a table of word-sized elements */
   asm  mov  ax,[SquareLookUpTable+bx];      /* look up the square */
   return(_AX);                               /* return the result */
}
```

Since *SquareLookUpTable* is in Turbo C's code segment, it would seem that a CS: segment override prefix should be required in order to read from it. In fact, this code automatically assembles with a CS: prefix on the access to *SquareLookUpTable;* Turbo C generates the correct assembler code to let Turbo Assembler know which segment *SquareLookUpTable* is in, and Turbo Assembler then generates segment override prefixes as needed.

Use the –1 switch for 80186/80286 instructions

If you want to use assembler instructions unique to the 80186 processor, such as

```
shr  ax,3
```

and

```
push  1
```

it's easiest to use the **–1** command-line option to Turbo C, as in this example,

```
tcc -1 -B heapmgr
```

where HEAPMGR.C is a program that contains inline assembly instructions unique to the 80186.

The primary purpose of the **–1** option is to instruct Turbo C to take advantage of the full 80186 instruction set when compiling, but the **–1** option also causes Turbo C to insert the **.186** directive at the start of the output assembler file; this instructs Turbo Assembler to assemble the full 80186 instruction set. Without the **.186** directive, Turbo Assembler will flag inline assembly instructions unique to the 80186 as errors. If you want to assemble 80186 instructions without having Turbo C use the full 80186 instruction set, just insert the line

```
asm  .186;
```

at the start of each Turbo C module containing inline 80186 instructions. This line will be passed through to the assembler file, where it will instruct Turbo Assembler to assemble 80186 instructions.

While Turbo C provides no built-in support for 80386, 80287, and 80387 processors, inline assembly that supports the 80286, 80287, 80386, and 80387 can be enabled in a similar manner, with the **asm** keyword and the **.286, .286C, .286P, .386, .386C, .386C, .287,** and **.387** Turbo Assembler directives.

The line

```
asm  .186;
```

illustrates an important point about inline assembly: *Any* valid assembler line can be passed to the assembler file by use of the **asm** prefix, including segment directives, equates, macros, and so on.

The format of inline assembly statements

Inline assembly statements are much like normal assembler lines, but there are a few differences. The format of an inline assembly statement is

```
asm  [<label>] <instruction/directive> <operands> <; or newline>
```

where

See "Memory and address operand limitations" on page 274 for important information regarding label.

- The **asm** keyword must start every inline assembly statement.
- [*<label>*] is a valid assembler label. The square brackets indicate that *label* is optional, just as it is in assembler.
- *<instruction/directive>* is any valid assembler instruction or directive.
- *<operands>* contains the operand(s) acceptable to the instruction or directive; it can also reference C constants, variables, and labels within the limitations described in the section "Limitations of inline assembly" on page 274.
- *<; or newline>* is a semicolon or a newline, either of which signals the end of the **asm** statement.

Semicolons in inline assembly

One aspect of inline assembly that no C purist could miss is that, alone among C statements, inline assembly statements do not require a terminating semicolon. A semicolon *can* be used to terminate each statement, but the end of the line will do just as well. So, unless you're planning to put multiple inline assembly statements on each line (which is not a good practice from the perspective of clarity), semicolons are purely optional. While this may not seem to be in the spirit of C, it is in keeping with the convention adopted by several UNIX-based compilers.

Comments in inline assembly

The previous description of the format of an inline assembly statement lacks one key element—a comment field. While semicolons can be placed at the end of inline assembly statements, semicolons do not begin comment fields in inline assembly code.

How, then, are you to comment your inline assembly code? Strangely enough, with C comments. Actually, that's not strange at all, for the C preprocessor processes inline assembly code along with the rest of your C code. This has the advantage of allowing you to use a uniform commenting style throughout your C programs containing inline assembly, and also makes it possible to use C-defined symbolic names in both C and inline assembly code. For example, in

```
     . . .
#define CONSTANT 51
     int i;
     . . .
     i = CONSTANT;                    /* set i to constant value */
     asm  sub  WORD PTR i,CONSTANT;  /* subtract const value from i */
     . . .
```

both C and inline assembly code use the C-defined symbol *CONSTANT*, and *i* winds up equal to 0.

The last example illustrates one wonderful feature of inline assembly, which is that the operand field might contain direct references not only to C-defined symbolic names but also to C variables. As you will see later in this chapter, accessing C variables in assembler is normally a messy task, and convenient reference to C variables is a primary reason why inline assembler is the preferred way to integrate assembler and C for most applications.

Accessing structure/union elements

Inline assembly code can directly reference structure elements. For example,

```
   . . .
struct Student {
   char Teacher[30];
   int Grade;
} JohnQPublic;
   . . .
   asm  mov  ax,JohnQPublic.Grade;
   . . .
```

loads AX with the contents of member *Grade* of the *Student* type structure *JohnQPublic*.

Inline assembly code can also access structure elements addressed relative to a base or index register. For instance,

```
. . .
asm  mov  bx,OFFSET JohnQPublic;
asm  mov  ax,[bx].Grade;
```

```
   . . .
```

also loads AX with member *Grade* of *JohnQPublic*. Since *Grade* is at offset 30 in the *Student* structure, the last example actually becomes

```
   . . .
   asm  mov   bx,OFFSET JohnQPublic;
   asm  mov   ax,[bx]+30
   . . .
```

The ability to access structure elements relative to a pointer register is very powerful, since it allows inline assembly code to handle arrays of structures and passed pointers to structures.

If, however, two or more structures that you're accessing with inline assembly code have the same member name, you must insert the following:

```
asm mov  bx,[di].(struct tm) tm_hour > alt
```

For example,

```
   . . .
struct Student {
   char Teacher[30];
   int Grade;
} JohnQPublic;
   . . .
struct Teacher {
   int Grade;
   long Income;
};
   . . .
asm  mov   ax,JohnQPublic.(struct Student) Grade
   . . .
```

An example of inline assembly

So far, you've seen a variety of code fragments that use inline assembly, but no real working inline assembly programs. This section remedies that situation by presenting a program that employs inline assembly to greatly speed the process of converting text to uppercase. The code presented in this section serves both as an example of what inline assembly can do and as a template to which you can refer to as you develop your own inline assembly code.

Take a moment to examine the programming problem to be solved by the sample program. We'd like to develop a function, named *StringToUpper*, that copies one string to another string, converting all lowercase characters to uppercase in the process. We'd also like to have this function work equally well with all strings in all memory models. One good way to do this is to have far string pointers passed to the function, since pointers to near strings can always be cast to pointers to far strings, but the reverse is not always true.

Unfortunately, we run into a performance issue here. While Turbo C handles far pointers perfectly well, far pointer-handling in Turbo C is much slower than near pointer-handling. This isn't a shortcoming of Turbo C, but rather an unavoidable effect when programming the 8086 in a high-level language.

On the other hand, string and far pointer-handling is one area in which assembler excels. The logical solution, then, is to use inline assembly to handle the far pointers and string copying, while letting Turbo C take care of everything else. The following program, STRINGUP.C, does exactly that:

```
/* Program to demonstrate the use of StringToUpper(). It calls
   StringToUpper to convert TestString to uppercase in Upper-
   CaseString, then prints UpperCaseString and its length. */

#pragma inline
#include <stdio.h>

/* Function prototype for StringToUpper() */
extern unsigned int StringToUpper(
unsigned char far * DestFarString,
unsigned char far * SourceFarString);

#define MAX_STRING_LENGTH 100

char *TestString = "This Started Out As Lowercase!";

char UpperCaseString[MAX_STRING_LENGTH];

main()
{
   unsigned int StringLength;

   /* Copy an uppercase version of TestString
      to UpperCaseString */
   StringLength = StringToUpper(UpperCaseString, TestString);

   /* Display the results of the conversion */
   printf("Original string:\n%s\n\n", TestString);
   printf("Uppercase string:\n%s\n\n", UpperCaseString);
   printf("Number of characters: %d\n\n", StringLength);
```

```
}

/* Function to perform high-speed translation to uppercase from
   one far string to another

   Input:
        DestFarString   - array in which to store uppercased
                          string (will be zero-terminated)
        SourceFarString - string containing characters to be
                          converted to all uppercase (must be
                          zero-terminated)

   Returns:
        The length of the source string in characters, not
        counting the terminating zero. */
unsigned int StringToUpper(unsigned char far * DestFarString,
                           unsigned char far * SourceFarString)
{
   unsigned int  CharacterCount;

   #define LOWER_CASE_A 'a'
   #define LOWER_CASE_Z 'z'
      asm ADJUST_VALUE  EQU  20h;        /* amount to subtract from
                                          lowercase letters to make
                                                 them uppercase */

      asm  cld;
      asm  push ds;                        /* save C's data segment */
      asm  lds  si,SourceFarString;          /* load far pointer to
                                                   source string */
      asm  les  di,DestFarString;            /* load far pointer to
                                              destination string */
      CharacterCount = 0;                    /* count of characters */
   StringToUpperLoop:
      asm  lodsb;                         /* get the next character */
      asm  cmp  al,LOWER_CASE_A;          /* if < a then it's not a
                                                 lowercase letter */
      asm  jb   SaveCharacter;
      asm  cmp  al,LOWER_CASE_Z;          /* if > z then it's not a
                                                 lowercase letter */
      asm  ja   SaveCharacter;
      asm  sub  al,ADJUST_VALUE;         /* it's lowercase; make it
                                                        uppercase */
   SaveCharacter:
      asm  stosb;                             /* save the character */
      CharacterCount++;                     /* count this character */
      asm  and  al,al;                     /* is this the ending 0? */
      asm  jnz  StringToUpperLoop;          /* no, process the next,
                                                     char, if any */
      CharacterCount--;            /* don't count the terminating 0 */
      asm  pop  ds;                     /* restore C's data segment */
      return(CharacterCount);
```

```
}
```

When run, STRINGUP.C displays the output

```
Original string:
This Started Out As Lowercase!

Uppercase string:
THIS STARTED OUT AS LOWERCASE!

Number of characters: 30
```

demonstrating that it does indeed convert all lowercase letters to uppercase.

The heart of STRINGUP.C is the function *StringToUpper*, which performs the entire process of string copying and conversion to uppercase. *StringToUpper* is written in both C and inline assembly, and accepts two far pointers as parameters. One far pointer points to a string containing text; the other far pointer points to another string, to which the text in the first string is to be copied with all lowercase letters converted to uppercase. The function declaration and parameter definition are all handled in C, and, indeed, a function prototype for *StringToUpper* appears at the start of the program. The main program calls *StringToUpper* just as if it were written in pure C. In short, all the advantages of programming in Turbo C are available, even though *StringToUpper* contains inline assembly code.

The body of *StringToUpper* is written in a mixture of C and inline assembly. Assembler is used to read each character from the source string, to check and, if need be, translate the character to uppercase, and to write the character to the destination string. Inline assembly allows *StringToUpper* to use the powerful **LODSB** and **STOSB** string instructions to read and write the characters.

In writing *StringToUpper*, we knew that we wouldn't need to access any data in Turbo C's data segment, so we simply pushed DS at the start of the function, then set DS to point to the source string and left it there for the rest of the function. One great advantage that inline assembly has over a pure C implementation is this ability to load the far pointers once at the start of the function and then never reload them until the function is done. By contrast, Turbo C and other high-level languages generally reload far pointers every time they are used. The ability to load far pointers just once means that *StringToUpper* processes far strings as rapidly as if they were near strings.

One other interesting point about *StringToUpper* is the way in which C and assembler statements are mixed. **#define** is used to set *LOWER_CASE_A* and *LOWER_CASE_Z*, while the assembler **EQU** directive is used to set *ADJUST_VALUE*, but all three symbols are used in the same fashion by the inline assembly code. Substitution for the C-defined symbols is done by the Turbo C preprocessor, while substitution for *ADJUST_VALUE* is done by Turbo Assembler, but both can be used by inline assembly code.

C statements to manipulate *CharacterCount* are sprinkled throughout *StringToUpper*. This was done only to illustrate that C code and inline assembly code can be intermixed. *CharacterCount* could just as easily have been maintained directly by inline assembly code in a free register, such as CX or DX; *StringToUpper* would then have run faster.

Freely intermixing C code and inline assembly code carries risks if you don't understand exactly what code Turbo C generates in between your inline assembly statements. Using the Turbo C's **-S** compiler option is the best way to explore what happens when you mix inline assembly and C code. For instance, you can learn exactly how the C and inline assembly code in *StringToUpper* fit together by compiling STRINGUP.C with the **-S** option and examining the output file STRINGUP.ASM.

STRINGUP.C vividly demonstrates the excellent payback that judicious use of inline assembly provides. In *StringToUpper*, the insertion of just 15 inline assembly statements approximately doubles string-handling speed over equivalent C code.

Limitations of inline assembly

There are very few limitations as to how inline assembly might be used; by and large, inline assembly statements are simply passed through to Turbo Assembler unchanged. There are, however, notable limitations involving certain memory and address operands, and a few other restrictions concerning register usage rules and the lack of default sizing of automatic C variables used in inline assembly.

Memory and address operand limitations

The only alterations Turbo C makes to inline assembly statements is to convert memory and memory address references, such as variable names and jump destinations, from their C representations to the assembler equivalents. These alterations introduce two limitations: Inline assembly jump instructions can only reference C labels, while inline assembly non-jump instructions can reference anything *but* C labels. For example,

```
  . . .
  asm  jz  NoDec;
  asm  dec  cx;
NoDec:
  . . .
```

is fine, but

```
. . .
asm  jnz  NoDec;
asm  dec  cx;
asm  NoDec:
. . .
```

will not compile properly. Similarly, inline assembly jumps cannot have function names as operands. Inline assembly instructions other than jumps can have any operands except C labels. For example,

```
  . . .
  asm  BaseValue  DB  '0';
  . . .
  asm  mov  al,BYTE PTR BaseValue;
  . . .
```

compiles, but

```
  . . .
BaseValue:
  asm  DB  '0';
  . . .
  asm  mov  al,BYTE PTR BaseValue;
  . . .
```

does not compile. Note that a call is not considered a jump, so valid operands to inline assembly calls include C function names and assembler labels, but not C labels. If a C function name is referenced in inline assembly code, it must be prefixed with an underscore; see the section "Underscores" on page 290 for details.

Lack of default automatic variable sizing in inline assembly

When Turbo C replaces a reference to an automatic variable in an inline assembly statement with an operand like *[BP-02]*, it does not place a size operator, such as **WORD PTR** or **BYTE PTR**, into the altered statement. This means that

```
. . .
int i;
. . .
asm  mov  ax,i;
. . .
```

is output to the assembler file as

```
mov  ax,[bp-02]
```

In this case, there's no problem, since the use of AX tells Turbo Assembler that this is a 16-bit memory reference. Moreover, the lack of a size operator gives you complete flexibility in controlling operand size in inline assembly. However, consider

```
. . .
int i;
. . .
asm  mov  i,0;
asm  inc  i;
. . .
```

which becomes

```
mov  [bp-02],0
inc  [bp-02]
```

Neither of these instructions has an inherent size, so Turbo Assembler can't assemble them. Consequently, when you refer to an automatic variable in Turbo Assembler without a register as either the source or the destination, be sure to use a size operator. The last example works just fine as

```
. . .
int i;
. . .
asm  mov  WORD PTR i,0;
asm  inc  BYTE PTR i;
. . .
```

The need to preserve registers

At the end of any inline assembly code you write, the following registers *must* contain the same values as they did at the start of the inline code: BP, SP, CS, DS, and SS. Failure to observe this rule can result in frequent program crashes and system reboots. AX, BX, CX, DX, SI, DI, ES, and the flags may be freely altered by inline code.

Preserving calling functions and register variables

Turbo C requires that SI and DI, which are used as register variables, not be destroyed by function calls. Happily, you don't have to worry about explicitly preserving SI or DI if you use them in inline assembly code. If Turbo C detects any use of those registers in inline assembly, it preserves them at the start of the function and restores them at the end—yet another of the conveniences of using inline assembly.

Suppressing internal register variables

Since register variables are stored in SI and DI, there would seem to be the potential for conflict between register variables in a given module and inline assembly code that uses SI or DI in that same module. Again, though, Turbo C anticipates this problem; any use of SI or DI in inline code will disable the use of that register to store register variables.

Turbo C version 1.0 did not guarantee avoidance of conflict between register variables and inline assembly code. If you are using version 1.0, you should either explicitly preserve SI and DI before using them in inline code or update to the latest version of the compiler.

Disadvantages of inline assembly versus pure C

We've spent a good bit of time exploring how inline assembly works and learning about the potential benefits of inline assembly. While inline assembly is a splendid feature for many applications, it does have certain disadvantages. Let's review those disadvantages, so you can make informed decisions about when to use inline assembly in your programs.

Reduced portability and maintainability

The very thing that makes inline assembly code so effective—the ability to program the 8086 processor directly—also detracts from a primary strength of C, portability. If you use inline assembly, it's a pretty safe bet that you won't be able to port your code to another processor or C compiler without changes.

Similarly, inline assembly code lacks the clear and concise formatting C provides, and is often unstructured as well. Consequently, inline assembly code is generally more difficult to read and maintain than C code.

When you use inline assembly code, it's a good practice to isolate the inline code in self-contained modules, and to structure the inline code carefully with plenty of comments. That way, it's easy to maintain the code, and it's a relatively simple matter to find the inline assembly code and rewrite it in C if you need to port the program to a different environment.

Slower compilation

Compilation of C modules containing inline assembly code is considerably slower than compilation of pure C code, primarily because inline assembly code must effectively be compiled twice, first by Turbo C and then again by Turbo Assembler. If Turbo C has to restart compilation because neither the **–B** option, the **–S** option, nor **#pragma inline** was used, compilation time for inline assembly becomes longer still. Fortunately, slow compilation of modules containing inline assembly is less of a problem now than it was in the past, since Turbo Assembler is so much faster than earlier assemblers.

Available with TCC only

As we mentioned earlier, the inline assembly feature is unique to TCC.EXE, the command-line version of Turbo C. TC.EXE, the integrated development environment version of Turbo C, does not support inline assembly.

Optimization loss

When inline assembly is used, Turbo C loses some control over the code of your programs, since you can directly insert any assembler statements into any C code. To some extent, you, as the inline assembly programmer, must compensate for this, by avoiding certain disruptive actions, such as failing to preserve the DS register or writing to the wrong area of memory.

On the other hand, Turbo C doesn't require you to follow all its internal rules when you program in inline assembler; if it did, you'd scarcely be better off using inline assembly than if you programmed in C and let Turbo C generate the code. What Turbo C does do is turn off some of its optimizations in functions containing inline assembly statements, thereby allowing you a relatively free hand in coding inline assembly. For example, some portions of the jump optimizer are turned off when inline assembly is used, and register variables are disabled if the inline code uses SI and DI. This partial loss of optimization is worth considering, given that you are presumably using inline assembly in order to boost code quality to its maximum.

If you are greatly concerned about producing the fastest or most compact code with inline assembly, you might want to write your functions that contain inline assembly code entirely in inline assembly—that is, don't mix C and inline assembly code within the same function. That way, you have control of the code in the inline assembly functions, Turbo C has control of the code in the C functions, and both you and Turbo C are free to generate the best possible code without restrictions.

Error trace-back limitations

Since Turbo C does little error-checking of inline assembly statements, errors in inline assembly code are often detected by Turbo Assembler, not Turbo C. Unfortunately, it can sometimes be difficult to relate the error messages produced by Turbo Assembler back to the original C source code, since the error messages and the line numbers they display are based on the .ASM file output by Turbo C and not the C code itself.

For example, in the course of compiling TEST.C, a C program containing inline assembly code, Turbo Assembler might complain about an incorrectly sized operand on line 23; unfortunately, "23" refers to the number of the error-producing line in TEST.ASM, the intermediate assembler file Turbo C generated for Turbo Assembler to assemble. You're on your own when it comes to figuring out what line in TEST.C is ultimately responsible for the error.

Your best bet in a case like this is to first locate the line causing the error in the intermediate .ASM file, which is left on the disk by Turbo C whenever Turbo Assembler reports assembly errors. The .ASM file contains special comments that identify the line in the C source file from which each block of assembler statements was generated; for example, the assembler lines following

```
; ?debug  L 15
```

were generated from line 15 of the C source file. Once you've located the line that caused the error in the .ASM file, you can then use the line-number comments to map the error-generating line back to the C source file.

Debugging limitations

Versions of Turbo C up to and including version 1.5 can't generate source-level debugging information (information required to let you see C source code as you debug) for modules containing inline assembly code. When inline assembly is used, Turbo C versions 1.5 and earlier generate plain assembler code with no embedded debugging information. Source-level debugging capabilities are lost, and only assembler-level debugging of C modules containing inline code is possible.

Later versions of Turbo C take advantage of special Turbo Assembler features to provide state-of-the-art, source-level debugging when used with Turbo Debugger to debug modules containing inline assembly code (and pure C modules too, of course).

Develop in C and compile the final code with inline assembly

In light of the disadvantages of inline assembly we've just discussed, it may seem that inline assembly should be used as sparingly as possible. Not so. The trick is to use inline assembly at the right point in the development cycle—at the end.

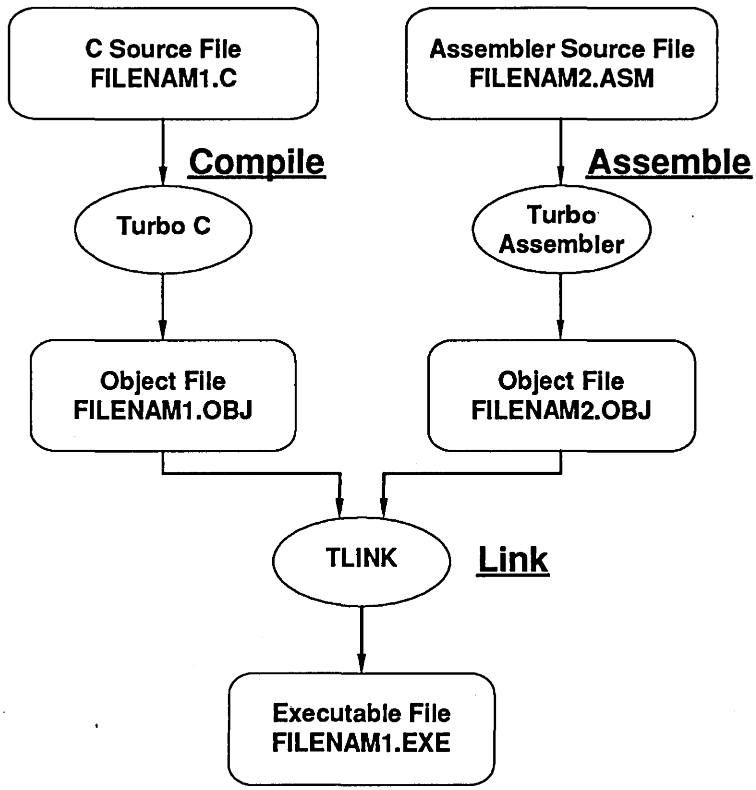
Most of the disadvantages of inline assembly boil down to a single problem: Inline assembly can slow down the edit/compile/debug cycle considerably. Slower compilation, inability to use the integrated environment, and difficulty in finding compilation errors all mean that development of code containing inline assembly statements will probably be slower than development of pure C code. Still, the proper use of inline assembly can result in dramatic improvements in code quality. What to do?

The answer is simple. Initially, develop each program entirely in C, taking full advantage of the excellent development environment provided by TC.EXE. When a program reaches full functionality, with the code debugged and running smoothly, switch to TCC.EXE and begin to convert critical portions of the program to inline assembly code. This approach allows you to develop and debug your overall program efficiently, then isolate and enhance selected sections of the code when it comes time to fine-tune the program.

Calling Turbo Assembler functions from Turbo C

C and assembler have traditionally been mixed by writing separate modules entirely in C or assembler, compiling the C modules and assembling the assembler modules, and then linking the separately compiled modules together. Turbo C modules can readily be linked with Turbo Assembler modules in this fashion. Figure 7.3 shows how to do this.

Figure 7.3
Compile, assemble, and link with Turbo C, Turbo Assembler, and TLINK



The executable file is produced from mixed C and assembler source files. You start this cycle with

```
tcc filenam1 filenam2.asm
```

This instructs Turbo C to first compile FILENAM1.C to FILENAM1.OBJ, then invoke Turbo Assembler to assemble FILENAM2.ASM to FILENAM2.OBJ, and finally invoke TLINK to link FILENAM1.OBJ and FILENAM2.OBJ into FILENAM1.EXE.

Separate compilation is very useful for programs that have sizable amounts of assembler code, since it makes the full power of Turbo Assembler available and allows you to do your assembly language programming in a pure assembler environment, without the **asm** keywords, extra compilation time, and C-related overhead of inline assembly.

There is a price to be paid for separate compilation: The assembler programmer must attend to all the details of interfacing C and assembler code. Where Turbo C handles segment specification, parameter-passing, reference to C variables, register variable preservation, and the like for inline assembly, separately compiled assembler functions must explicitly do all that and more.

There are two major aspects to interfacing Turbo C and Turbo Assembler. First, the various parts of the C and assembler code must be linked together properly, and functions and variables in each part of the code must be made available to the rest of the code as needed. Second, the assembler code must properly handle C-style function calls. This includes accessing passed parameters, returning values, and following the register preservation rules required of C functions.

Let's start by examining the rules for linking together Turbo C and Turbo Assembler code.

The framework

In order to link Turbo C and Turbo Assembler modules together, three things must happen:

- The Turbo Assembler modules must use a Turbo C-compatible segment-naming scheme.
- The Turbo C and Turbo Assembler modules must share appropriate function and variable names in a form acceptable to Turbo C.
- TLINK must be used to combine the modules into an executable program.

This says nothing about what the Turbo Assembler modules actually *do*; at this point, we're only concerned with creating a framework within which C-compatible Turbo Assembler functions can be written.

Memory models and segments

For a given assembler function to be callable from C, that function must use the same memory model as the C program and must use a C-compatible code segment. Likewise, in order for data defined in an assembler module to be accessed by C code (or for C data to be accessed by assembler code), the assembler code must follow C data segment-naming conventions.

See "Standard segment directives" in Chapter 5, page 111, for an introduction to the simplified segment directives.

Memory models and segment-handling can be quite complex to implement in assembler. Fortunately, Turbo Assembler does virtually all the work of implementing Turbo C-compatible memory models and segments for you in the form of the simplified segment directives.

Simplified segment directives and Turbo C

The **DOSSEG** directive instructs Turbo Assembler to order segments according to the Intel segment-ordering conventions, the same conventions followed by Turbo C (and many other popular language products, including those from Microsoft).

The **.MODEL** directive tells Turbo Assembler that segments created with the simplified segment directives should be compatible with the selected memory model (tiny, small, compact, medium, large, or huge), and controls the default type (near or far) of procedures created with the **PROC** directive. Memory models defined with the **.MODEL** directive are compatible with the equivalently named Turbo C models.

Finally, the **.CODE**, **.DATA**, **.DATA**, **.FARDATA**, **.FARDATA**, and **.CONST** simplified segment directives generate Turbo C-compatible segments.

For example, consider the following Turbo Assembler module, named DOTOTAL.ASM:

Underscores (_) prefix many of the labels in DoTotal because they are normally required by Turbo C. For more detail, see the section "Underscores" on page 290.

```
; select Intel-convention segment ordering
       .MODEL  small    ;select small model (near code and data)
       .DATA            ;TC-compatible initialized data segment
       EXTRN   _Repetitions:WORD   ;externally defined
       PUBLIC  _StartingValue      ;available to other modules
_StartingValue  DW  0
       .DATA?           ;TC-compatible uninitialized data segment
RunningTotal    DW  ?
       .CODE            ;TC-compatible code segment
       PUBLIC  _DoTotal
```

```
_DoTotal      PROC    ;function (near-callable in small model)
        mov     cx,[_Repetitions]     ;# of counts to do
        mov     ax,[_StartingValue]
        mov     [RunningTotal],ax     ;set initial value
TotalLoop:
        inc     [RunningTotal]        ;RunningTotal++
        loop    TotalLoop
        mov     ax,[RunningTotal]     ;return final total
        ret
_DoTotal      ENDP
        END
```

The assembler procedure *_DoTotal* is readily callable from a small-model Turbo C program with the statement

```
DoTotal();
```

Note that *_DoTotal* expects some other part of the program to define the external variable *Repetitions*. Similarly, the variable *StartingValue* is made public, so other portions of the program can access it. The following Turbo C module, SHOWTOT.C, accesses public data in DOTOTAL.ASM and provides external data to DOTOTAL.ASM:

```
extern int StartingValue;
extern int DoTotal(void);
int Repetitions;
main()
{
   int i;
   Repetitions = 10;
   StartingValue = 2;
   printf("%d\n", DoTotal());
}
```

To create the executable program SHOWTOT.EXE from SHOWTOT.C and DOTOTAL.ASM, enter the command line

```
tcc showtot dototal.asm
```

If you wanted to link *_DoTotal* to a compact-model C program, you would simply change the **.MODEL** directive to **.MODEL** *COMPACT*. If you wanted to use a far segment in DOTOTAL.ASM, you could use the **.FARDATA** directive.

In short, generating the correct segment ordering, memory model, and segment names for linking with Turbo C is a snap with the simplified segment directives.

Old-style segment directives and Turbo C

Simply put, it's a nuisance interfacing Turbo Assembler code to C code using the old-style segment directives. For example, if you replace the simplified segment directives in DOTOTAL.ASM with old-style segment directives, you get

```
DGROUP  GROUP    _DATA,_BSS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
        EXTRN    _Repetitions:WORD    ;externally defined
        PUBLIC   _StartingValue       ;available to other modules
_StartingValue    DW 0
_DATA   ENDS
_BSS    SEGMENT   WORD PUBLIC 'BSS'
RunningTotal      DW  ?
_BSS    ENDS
_TEXT   SEGMENT   BYTE PUBLIC 'CODE'
        ASSUME    cs:_TEXT,ds:DGROUP,ss:DGROUP
        PUBLIC    _DoTotal
_DoTotal          PROC                ;function (near-callable
                                      ; in small model)
        mov       cx,[_Repetitions]   ;# of counts to do
        mov       ax,[_StartingValue]
        mov       [RunningTotal],ax   ;set initial value
TotalLoop:
        inc       [RunningTotal]      ;RunningTotal++
        loop      TotalLoop
        mov       ax,[RunningTotal]   ;return final total
        ret
_DoTotal ENDP
_TEXT   ENDS
        END
```

For an overview of Turbo C segment usage, refer to Chapter 4 of the Turbo C Programmer's Guide.

The version with old-style segment directives is not only longer, but also much harder to read and harder to change to match a different C memory model. When you're interfacing to Turbo C, there's generally no advantage to using the old-style segment directives. If you still want to use the old-style segment directives when interfacing to Turbo C, you'll have to identify the correct segments for the memory model your C code uses.

The easiest way to determine the appropriate old-style segment directives for linking with a given Turbo C program is to compile the main module of the Turbo C program in the desired memory model with the **-S** option, which causes Turbo C to generate an assembler version of the C code. In that C code, you'll find all the old-style segment directives used by Turbo C; just copy them into your assembler code. For example, if you enter the command

```
tcc -S showtot.c
```

the file SHOWTOT.ASM is generated:

```
        ifndef  ??version
?debug  macro
        ENDM
        ENDIF
        NAME    showtot
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
DGROUP  GROUP   _DATA,_BSS
        ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT   ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_d@     LABEL  BYTE
_d@w    LABEL  WORD
_DATA   ENDS
_BSS    SEGMENT WORD PUBLIC 'BSS'
_b@     LABEL  BYTE
_b@w    LABEL  WORD
        ?debug  C E91481D5100973686F77746F742E63
_BSS    ENDS
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
;       ?debug  L 3
_main   PROC    NEAR
;       ?debug  L 6
        mov     WORD PTR DGROUP:_Repetitions,10
;       ?debug  L 7
        mov     WORD PTR DGROUP:_StartingValue,2
;       ?debug  L 8
        call    NEAR PTR _DoTotal
        push    ax
        mov     ax,offset DGROUP:_s@
        push    ax
        call    NEAR PTR _printf
        pop     cx
        pop     cx
@1:
;       ?debug  L 9
        ret
_main   ENDP
_TEXT   ENDS
_BSS    SEGMENT WORD PUBLIC 'BSS'
_Repetitions    LABEL   WORD
        DB      2 dup (?)
        ?debug  C E9
_BSS    ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_s@     LABEL  BYTE
```

```
        DB      37
        DB      100
        DB      10
        DB      0
_DATA   ENDS
        EXTRN   _StartingValue:WORD
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
        EXTRN   _DoTotal:NEAR
        EXTRN   _printf:NEAR
_TEXT   ENDS
        PUBLIC  _Repetitions
        PUBLIC  _main
        END
```

Chapter 9 covers segment directives in detail.

The segment directives for **_DATA** (the initialized data segment), **_TEXT** (the code segment), and **_BSS** (the uninitialized data segment), along with the **GROUP** and **ASSUME** directives, are in ready-to-assemble form, so you can use them as is.

Segment defaults: When is it necessary to load segments?

Under some circumstances, your C-callable assembler functions might have to load DS and/or ES in order to access data. It's also useful to know the relationships between the settings of the segment registers on a call from Turbo C, since sometimes assembler code can take advantage of the equivalence of two segment registers. Let's take a moment to examine the settings of the segment registers when an assembler function is called from Turbo C, the relationships between the segment registers, and the cases in which an assembler function might need to load one or more segment registers.

On entry to an assembler function from Turbo C, the CS and DS registers have the following settings, depending on the memory model in use (SS is always used for the stack segment, and ES is always used as a scratch segment register):

Table 7.1 Register settings when Turbo C enters assembler

Model	CS	DS
Tiny	_TEXT	DGROUP
Small	_TEXT	DGROUP
Compact	_TEXT	DGROUP
Medium	filename_TEXT	DGROUP
Large	filename_TEXT	DGROUP
Huge	filename_TEXT	calling_filename_DATA

filename is the name of the assembler module, and *calling_filename* is the name of the module calling the assembler module.

In the tiny model, **_TEXT** and **DGROUP** are the same, so CS equals DS on entry to functions. Also in the tiny, small, and medium models, SS equals DS on entry to functions.

So, when is it necessary to load a segment register in a C-callable assembler function? For starters, you should never have to (or want to) directly load the CS or SS registers. CS is automatically set as needed on far calls, jumps, and returns, and can't be tampered with otherwise. SS always points to the stack segment, which should never change during the course of a program (unless you're writing code that switches stacks, in which case you had best know *exactly* what you're doing!).

ES is always available for you to use as you wish. You can use ES to point at far data, or you can load ES with the destination segment for a string instruction.

That leaves the DS register. In all Turbo C models other than the huge model, DS points to the static data segment (**DGROUP**) on entry to functions, and that's generally where you'll want to leave it. You can always use ES to access far data, although you may find it desirable to instead temporarily point DS to far data that you're going to access intensively, thereby saving many segment override instructions in your code. For example, you could access a far segment in either of the following ways:

```
         . . .
         .FARDATA
Counter  DW      0
         . . .
         .CODE
         PUBLIC  _AsmFunction
_AsmFunction     PROC
         . . .
         mov     ax,@fardata
         mov     es,ax            ;point ES to far data segment
         inc     es:[Counter]     ;increment counter variable
         . . .
_AsmFunction     ENDP
         . . .
```

or

```
        . . .
        .FARDATA
Counter DW      0
        . . .
        .CODE
        PUBLIC  _AsmFunction
_AsmFunction    PROC
        . . .
        ASSUME  ds:@fardata
        mov     ax,@fardata
        mov     ds,ax               ;point DS to far data segment
        inc     [Counter]           ;increment counter variable
        ASSUME  ds:@data
        mov     ax,@data
        mov     ds,ax               ;point DS back to DGROUP
        . . .
_AsmFunction    ENDP
        . . .
```

The second version has the advantage of not requiring an ES: override on each memory access to the far data segment. If you do load DS to point to a far segment, be sure to restore it as in the preceding example before attempting to access any variables in **DGROUP**. Even if you don't access **DGROUP** in a given assembler function, be sure to restore DS before exiting, since Turbo C assumes that functions leave DS unchanged.

Handling DS in C-callable huge model functions is a bit different. In the huge model, Turbo C doesn't use **DGROUP** at all. Instead, each module has its own data segment, which is a far segment relative to all the other modules in the program; there is no commonly shared near data segment. On entry to a function in the huge model, DS should be set to point to that module's far segment and left there for the remainder of the function, as follows:

```
        . . .
        .FARDATA
        . . .
        .CODE
        PUBLIC  _AsmFunction
_AsmFunction    PROC
        push    ds
        mov     ax,@fardata
        mov     ds,ax
        . . .
        pop     ds
        ret
```

```
_AsmFunction     ENDP
        . . .
```

Note that the original state of DS is preserved with a **PUSH** on entry to *AsmFunction* and restored with a **POP** before exiting; even in the huge model, Turbo C requires all functions to preserve DS.

Publics and externals

Turbo Assembler code can call C functions and reference external C variables, and Turbo C code can likewise call public Turbo Assembler functions and reference public Turbo Assembler variables. Once Turbo C-compatible segments are set up in Turbo Assembler, as described in the preceding sections, only the following few simple rules need be observed in order to share functions and variables between Turbo C and Turbo Assembler.

Underscores

Normally, Turbo C expects all external labels to start with an underscore character (_). Turbo C automatically prefixes an underscore to all function and external variable names when they're used in C code, so you only need to attend to underscores in your assembler code. You must be sure that all assembler references to Turbo C functions and variables begin with underscores, and you must begin all assembler functions and variables that are made public and referenced by Turbo C code with underscores.

For example, the following C code,

```
extern int ToggleFlag();
int Flag;
main()
{
   ToggleFlag();
}
```

links properly with the following assembler program:

```
        .MODEL  small
        .DATA
        EXTRN   _Flag:WORD
        .CODE
        PUBLIC  _ToggleFlag
_ToggleFlag     PROC
        cmp     [_Flag],0               ;is the flag reset?
        jz      SetFlag                 ;yes, set it
        mov     [_Flag],0               ;no, reset it
```

Labels not referenced by C code, such as SetFlag, don't need leading underscores.

```
        jmp     short EndToggleFlag     ;done
SetFlag:
        mov     [_Flag],1               ;set flag
EndToggleFlag:
        ret
_ToggleFlag     ENDP
        END
```

When you use the C language specifier in your **EXTRN** and **PUBLIC** directives,

```
     DOSSEG
     .MODEL    SMALL
     .DATA
     EXTRN     C Flag:word
     .CODE
     PUBLIC    C ToggleFlag
ToggleFlag     PROC
     cmp       [Flag],0
     jz        SetFlag
     mov       [Flag],0
     jmp       short EndToggleFlag
SetFlag:
     mov       [Flag],1
EndToggleFlag:
     ret
ToggleFlag     ENDP
     END
```

Turbo Assembler causes the underscores to be prefixed automatically when *Flag* and *ToggleFlag* are published in the object module.

By the way, it *is* possible to tell Turbo C not to use underscores by using the **–u-** command-line option. But you have to purchase the run-time library source from Borland and recompile the libraries with underscores disabled in order to use the **–u-** option. (See "Pascal calling conventions" on page 307 for information on the **–p** option, which disables the use of underscores and case-sensitivity.)

The significance of uppercase and lowercase

Turbo Assembler is normally insensitive to case when handling symbolic names, making no distinction between uppercase and lowercase letters. Since C is case-sensitive, it's desirable to have Turbo Assembler be case-sensitive, at least for those symbols that are shared between assembler and C. **/ml** and **/mx** make this possible.

The **/ml** command-line switch causes Turbo Assembler to become case-sensitive for all symbols. The **/mx** command-line switch causes Turbo Assembler to become case-sensitive for public (**PUBLIC**), external (**EXTRN**), global (**GLOBAL**), and communal (**COMM**) symbols only.

Label types

While assembler programs are free to access any variable as data of any size (8 bit, 16 bit, 32 bit, and so on), it is generally a good idea to access variables in their native size. For instance, it usually causes problems if you write a word to a byte variable:

```
        . . .
SmallCount  DB  0
        . . .
        mov  WORD PTR [SmallCount],0ffffh
        . . .
```

Consequently, it's important that your assembler **EXTRN** statements that declare external C variables specify the right size for those variables, since Turbo Assembler has only your declaration to go by when deciding what size access to generate to a C variable. Given the statement

```
char c
```

in a C program, the assembler code

```
        . . .
EXTRN  c:WORD
        . . .
inc    [c]
        . . .
```

could lead to nasty problems, since every 256th time the assembler code incremented *c*, *c* would turn over. And, since *c* is erroneously declared as a word variable, the byte at **OFFSET** c + 1 would incorrectly be incremented, with unpredictable results.

Correspondence between C and assembler data types is as follows:

C Data Type	Assembler Data Type
unsigned char	byte
char	byte
enum	word
unsigned short	word
short	word
unsigned int	word
int	word
unsigned long	dword
long	dword
float	dword
double	qword
long double	tbyte
near *	word
far *	dword

Far externals

If you're using the simplified segment directives, **EXTRN** declarations of symbols in far segments must not be placed within any segment, since Turbo Assembler considers symbols declared within a given segment to be associated with that segment. This has its drawbacks: Turbo Assembler cannot check the addressability of symbols declared **EXTRN** outside any segment, and so can neither generate segment overrides as needed nor inform you when you attempt to access that variable when the correct segment is not loaded. Turbo Assembler still assembles the correct code for references to such external symbols, but can no longer provide the normal degree of segment addressability checking.

If you want to (though we discourage it), you can use the old-style segment directives to explicitly declare the segment each external symbol is in and then place the **EXTRN** directive for that symbol inside the segment declaration. However, this is a good bit of work; if you don't mind taking responsibility for making sure that the correct segment is loaded when you access far data, it's easiest to just put **EXTRN** declarations of far symbols outside all segments. For example, suppose that FILE1.ASM contains

```
   . . .
        .FARDATA
File1Variable   DB   0
   . . .
```

Then if FILE1.ASM is linked to FILE2.ASM, which contains

```
   . . .
        .DATA
        EXTRN  File1Variable:BYTE
        .CODE
Start   PROC
        mov    ax,SEG File1Variable
        mov    ds,ax
   . . .
```

SEG *File1Variable* will not return the correct segment. The **EXTRN** directive is placed within the scope of the **DATA** directive of FILE2.ASM, so Turbo Assembler considers *File1Variable* to be in the near **DATA** segment of FILE2.ASM, rather than in the **FARDATA** segment.

The following code for FILE2.ASM allows **SEG** *File1Variable* to return the correct segment:

```
   . . .
        .DATA
@curseg ENDS
        EXTRN   File1Variable:BYTE
        .CODE
Start   PROC
        mov     ax,SEG File1Variable
        mov     ds,ax
   . . .
```

The trick here is that the **@curseg ENDS** directive ends the **.DATA** segment, so no segment directive is in effect when *File1Variable* is declared external.

Linker command line

The simplest way to link Turbo C modules with Turbo Assembler modules is to enter a single Turbo C command line and let Turbo C do all the work. Given the proper command line, Turbo C will compile the C code, invoke Turbo Assembler to do the assembling, and invoke TLINK to link the object files into an executable file. Suppose, for example, that you have a program consisting of the C files MAIN.C and STAT.C and the assembler files SUMM.ASM and DISPLAY.ASM. The command line

```
tcc main stat summ.asm display.asm
```

compiles MAIN.C and STAT.C, assembles SUMM.ASM and DISPLAY.ASM, and links all four object files, along with the C start-up code and any required library functions, into MAIN.EXE. You only need remember the .ASM extensions when typing your assembler file names.

If you use TLINK in stand-alone mode, the object files generated by Turbo Assembler are standard object modules and are treated just like C object modules.

Between Turbo Assembler and Turbo C

Now that you understand how to build and link C-compatible assembler modules, you need to learn what sort of code you can put into C-callable assembler functions. There are three areas to examine here: receiving passed parameters, using registers, and returning values to the calling code.

Parameter-passing

Turbo C passes parameters to functions on the stack. Before calling a function, Turbo C first pushes the parameters to that function onto the stack, starting with the rightmost parameter and ending with the leftmost parameter. The C function call

```
. . .
Test(i, j, 1);
. . .
```

compiles to

```
mov   ax,1
push  ax
push  WORD PTR DGROUP:_j
push  WORD PTR DGROUP:_i
call  NEAR PTR _Test
add   sp,6
```

in which you can clearly see the rightmost parameter, 1, being pushed first, then *j*, and finally *i*.

Read about Pascal calling conventions on page 307.

Upon return from a function, the parameters that were pushed on the stack are still there, but are no longer of any use. Consequently, immediately following each function call, Turbo C adjusts the stack pointer back to the value it contained before the parameters were pushed, thereby discarding the parameters. In the previous example, the three parameters of 2 bytes each take up 6 bytes of stack space altogether, so Turbo C adds 6 to the stack pointer to discard the parameters after the call to *Test*. The important point here is that under C calling conventions, the *calling* code is responsible for discarding the parameters from the stack.

Assembler functions can access parameters passed on the stack relative to the BP register. For example, suppose the function *Test* in the previous example is the following assembler function:

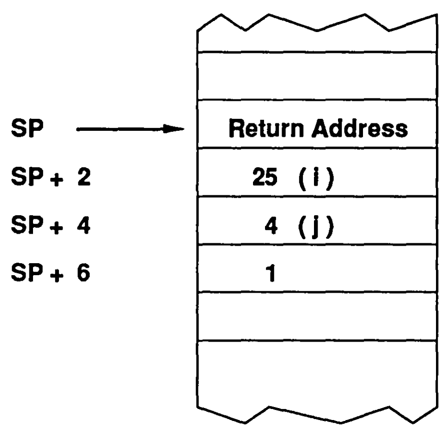
```
        .MODEL  small
        .CODE
        PUBLIC  _Test
_Test   PROC
        push    bp
        mov     bp,sp
        mov     ax,[bp+4]           ;get parameter 1
        add     ax,[bp+6]           ;add parameter 2 to parameter 1
        sub     ax,[bp+8]           ;subtract parameter 3 from sum
        pop     bp
        ret
_Test   ENDP
        END
```

You can see that *Test* is getting the parameters passed by the C code from the stack, relative to BP. (Remember that BP addresses the stack segment.) But just how are you to know *where* to find the parameters relative to BP?

Figure 7.4 shows what the stack looks like just before the first instruction in *Test* is executed:

```
i = 25;
j = 4;
Test(i, j, 1);
```

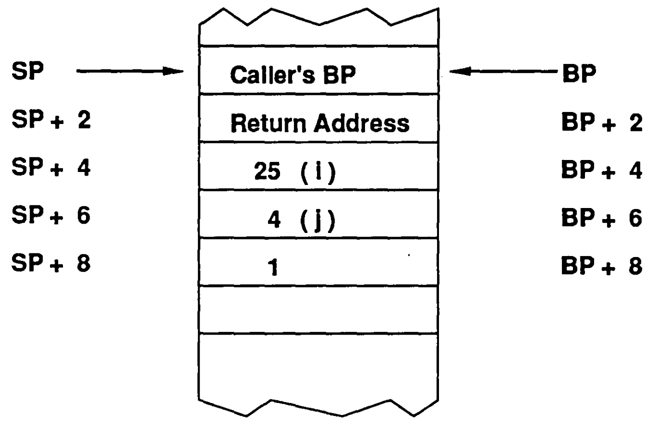
Figure 7.4
State of the stack just before executing Test's first instruction



The parameters to *Test* are at fixed locations relative to SP, starting at the stack location 2 bytes higher than the location of the return address that was pushed by the call. After loading BP with SP, you can access the parameters relative to BP. However, you must first preserve BP, since the calling C code expects you to return with BP unchanged. Pushing BP changes all the offsets on the stack. Figure 7.5 shows the stack after these lines of code are executed:

```
 . . .
push  bp
mov   bp,sp
 . . .
```

Figure 7.5
State of the stack after PUSH and MOV



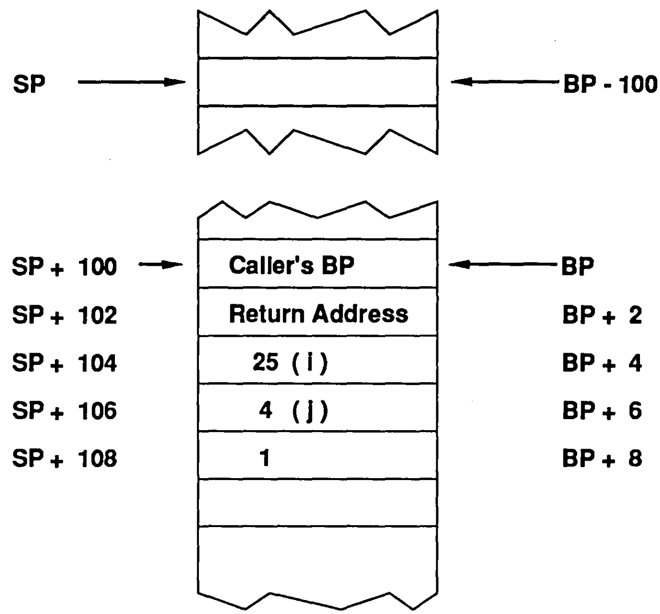
This is the standard C stack frame, the organization of a function's parameters and automatic variables on the stack. As you can see, no matter how many parameters a C program might have, the leftmost parameter is always stored at the stack address immediately above the pushed return address, the next parameter to the right is stored just above the leftmost parameter, and so on. As long as you know the order and type of the passed parameters, you always know where to find them on the stack.

Space for automatic variables can be reserved by subtracting the required number of bytes from SP. For example, room for a 100-byte automatic array could be reserved by starting *Test* with

```
 . . .
push  bp
mov   bp,sp
sub   sp,100
 . . .
```

as shown in Figure 7.6.

Figure 7.6
State of the stack after PUSH, MOV, and SUB



Since the portion of the stack holding automatic variables is at a lower address than BP, negative offsets from BP are used to address automatic variables. For example,

```
mov  BYTE PTR [bp-100],0
```

would set the first byte of the 100-byte array you reserved earlier to zero. Passed parameters, on the other hand, are always addressed at positive offsets from BP.

While you can, if you wish, allocate space for automatic variables as shown previously, Turbo Assembler provides a special version of the **LOCAL** directive that makes allocation and naming of automatic variables a snap. When **LOCAL** is encountered within a procedure, it is assumed to define automatic variables for that procedure. For example,

```
LOCAL   LocalArray:BYTE:100,LocalCount:WORD = AUTO_SIZE
```

defines the automatic variables *LocalArray* and *LocalCount*. *LocalArray* is actually a label equated to *[BP-100]*, and *LocalCount* is actually a label equated to *[BP-102]*, but you can use them as variable names without ever needing to know their values. *AUTO_SIZE* is the total number of bytes of automatic storage required; you must subtract this value from SP in order to allocate space for the automatic variables.

Here's how you might use **LOCAL:**

```
     . . .
_TestSub  PROC
     LOCAL     LocalArray:BYTE:100,LocalCount:WORD=AUTO_SIZE
     push bp                ;preserve caller's stack frame pointer
     mov  bp,sp             ;set up our own stack frame pointer
     sub  sp,AUTO_SIZE      ;allocate room for automatic variables
     mov  [LocalCount],10   ;set local count variable to 10
                            ; (LocalCount is actually [BP-102])
     . . .
     mov  cx,[LocalCount]   ;get count from local variable
     mov  al,'A'            ;we'll fill with character "A"
     lea  bx,[LocalArray]   ;point to local array
                            ; (LocalArray is actually [BP-100])
FillLoop:
     mov  [bx],al           ;fill next byte
     inc  bx                ;point to following byte
     loop FillLoop          ;do next byte, if any
     mov  sp,bp             ;deallocate storage for automatic
                            ; variables (add sp,AUTO_SIZE would
                            ; also have worked)
     pop  bp                ;restore caller's stack frame pointer
     ret
_TestSub  ENDP
     . . .
```

In this example, note that the first field after the definition of a given automatic variable is the data type of the variable: **BYTE**, **WORD**, **DWORD**, **NEAR**, and so on. The second field after the definition of a given automatic variable is the number of elements of that variable's type to reserve for that variable. This field is optional and defines an automatic array if used; if it is omitted, one element of the specified type is reserved. Consequently, *LocalArray* consists of 100 byte-sized elements, while *LocalCount* consists of 1 word-sized element.

Also note that the **LOCAL** line in the preceding example ends with *=AUTO_SIZE*. This field, beginning with an equal sign, is optional; if present, it sets the label following the equal sign to the number of bytes of automatic storage required. You must then use that label to allocate and deallocate storage for automatic variables, since the **LCCAL** directive only generates labels, and doesn't actually generate any code or data storage. To put this another way: **LOCAL** doesn't allocate automatic variables, but simply generates labels that you can readily use to both allocate storage for and access automatic variables.

A very handy feature of **LOCAL** is that the labels for both the automatic variables and the total automatic variable size are limited in scope to the procedure they're used in, so you're free to reuse an automatic variable name in another procedure.

*Refer to Chapter 3 in the Reference Guide for additional information about both forms of the **LOCAL** directive.*

As you can see, **LOCAL** makes it much easier to define and use automatic variables. Note that the **LOCAL** directive has a completely different meaning when used in macros, as discussed in Chapter 9.

By the way, Turbo C handles stack frames in just the way we've described here. You may well find it instructive to compile a few Turbo C modules with the **–S** option and look at the assembler code Turbo C generates to see how Turbo C creates and uses stack frames.

So far, so good, but there are further complications. First of all, this business of accessing parameters at constant offsets from BP is a nuisance; not only is it easy to make mistakes, but if you add another parameter, all the other stack frame offsets in the function must be changed. For example, suppose you change *Test* to accept four parameters:

```
Test(Flag, i, j, 1);
```

Suddenly *i* is at offset 6, not offset 4, *j* is at offset 8, not offset 6, and so on. You can use equates for the parameter offsets:

```
        . . .
Flag         EQU  4
AddParm1     EQU  6
AddParm2     EQU  8
SubParm1     EQU  10

        mov  ax,[bp+AddParm1]
        add  ax,[bp+AddParm2]
        sub  ax,[bp+SubParm1]
        . . .
```

but it's still a nuisance to calculate the offsets and maintain them. There's a more serious problem, too: The size of the pushed return address grows by 2 bytes in far code models, as do the sizes of passed code pointers and data pointer in far code and far data models, respectively. Writing a function that can be easily assembled to access the stack frame properly in any memory model would thus seem to be a difficult task.

Fear not. Turbo Assembler provides you with the **ARG** directive, which makes it easy to handle passed parameters in your assembler routines.

The **ARG** directive automatically generates the correct stack offsets for the variables you specify. For example,

```
arg  FillArray:WORD,Count:WORD,FillValue:BYTE
```

specifies three parameters: *FillArray*, a word-sized parameter; *Count*, a word-sized parameter, and *FillValue*, a byte-sized parameter. **ARG** actually sets the label *FillArray* to [*BP+4*] (assuming the example code resides in a near procedure), the label *Count* to [*BP+6*], and the label *FillValue* to [*BP+8*]. However, **ARG** is valuable precisely because you can use **ARG**-defined labels without ever knowing the values they're set to.

For example, suppose you've got a function *FillSub*, called from C as follows:

```
main()
{
#define ARRAY_LENGTH 100
   char TestArray[ARRAY_LENGTH];

   FillSub(TestArray,ARRAY_LENGTH,'*');
}
```

You could use **ARG** in *FillSub* to handle the parameters as follows:

```
_FillSub  PROC NEAR
     ARG  FillArray:WORD,Count:WORD,FillValue:BYTE
     push bp                   ;preserve caller's stack frame
     mov  bp,sp                ;set our own stack frame
     mov  bx,[FillArray]       ;get pointer to array to fill
     mov  cx,[Count]           ;get length to fill
     mov  al,[FillValue]       ;get value to fill with
FillLoop:
     mov  [bx],al              ;fill a character
     inc  bx                   ;point to next character
     loop FillLoop             ;do next character
     pop  bp                   ;restore caller's stack frame
     ret
_FillSub  ENDP
```

*Look at Chapter 3 in the Reference Guide for additional information about the **ARG** directive.*

That's really all it takes to handle passed parameters with **ARG**. Better yet, **ARG** automatically accounts for the different sizes of near and far returns. Another convenience is that the labels defined with **ARG** are limited in scope to the procedure they're used in when you declare them using the local label prefix (see

LOCALS in the *Reference Guide*). So you need never worry about conflict between parameter names in different procedures.

Preserving registers

As far as Turbo C is concerned, C-callable assembler functions can do anything they please, as long as they preserve the following registers: BP, SP, CS, DS, and SS. While these registers can be altered during the course of an assembler function, when the calling code is returned, they must be exactly as they were when the assembler function was called. AX, BX, CX, DX, ES, and the flags can be changed in any way.

SI and DI are special cases, since they're used by Turbo C as register variables. If register variables are enabled in the C module calling your assembler function, you must preserve SI and DI; but if register variables are not enabled, SI and DI need not be preserved.

It's good practice to always preserve SI and DI in your C-callable assembler functions, regardless of whether register variables are enabled. You never know when you might link a given assembler module to a different C module, or recompile your C code with register variables enabled, without remembering that your assembler code needs to be changed as well.

Returning values

A C-callable assembler function can return a value, just like a C function. Function values are returned as follows:

Return Value Type	Return Value Location
unsigned char	AX
char	AX
enum	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX:AX
long	DX:AX
float	8087 top-of-stack (TOS) register (ST(0))
double	8087 top-of-stack (TOS) register (ST(0))
long double	8087 top-of-stack (TOS) register (ST(0))
near *	AX
far *	DX:AX

In general, 8- and 16-bit values are returned in AX, and 32-bit values are returned in DX:AX, with the high 16 bits of the value in

DX. Floating-point values are returned in ST(0), which is the 8087's top-of-stack (TOS) register, or in the 8087 emulator's TOS register if the floating-point emulator is being used.

Structures are a bit more complex. Structures that are 1 or 2 bytes in length are returned in AX, and structures that are 4 bytes in length are returned in DX:AX. Three-byte structures and structures larger than 4 bytes must be stored in a static data area, and a pointer to that static data must then be returned. As with all pointers, near pointers to structures are returned in AX, and far pointers to structures are returned in DX:AX.

Let's look at a small model C-callable assembler function, *FindLastChar*, that returns a pointer to the last character of a passed string. The C prototype for this function would be

```
extern char * FindLastChar(char * StringToScan);
```

where *StringToScan* is the nonempty string for which a pointer to the last character is to be returned.

Here's *FindLastChar*:

```
        .MODEL  small
        .CODE
        PUBLIC _FindLastChar
_FindLastChar   PROC
        push    bp
        mov     bp,sp
        cld             ;we need string instructions to count up
        mov     ax,ds
        mov     es,ax   ;set ES to point to the near data segment
        mov     di,     ;point ES:DI to start of passed string
        mov     al,0    ;search for the null that ends the string
        mov     cx,0ffffh  ;search up to 64K-1 bytes
        repnz   scasb      ;look for the null
        dec     di         ;point back to the null
        dec     di         ;point back to the last character
        mov     ax,di      ;return the near pointer in AX
        pop     bp
        ret
_FindLastChar   ENDP
        END
```

The final result, the near pointer to the last character in the passed string, is returned in AX.

Calling an assembler function from C

Now look at an example of Turbo C code calling a Turbo Assembler function. The following Turbo Assembler module, COUNT.ASM, contains the function *LineCount*, which returns counts of the number of lines and characters in a passed string:

```
; Small model C-callable assembler function to count the number
; of lines and characters in a zero-terminated string.
;
; Function prototype:
;      extern unsigned int LineCount(char * near StringToCount,
;             unsigned int near * CharacterCountPtr);
; Input:
;      char near * StringToCount: pointer to the string on which
;      a line count is to be performed
;
;      unsigned int near * CharacterCountPtr: pointer to the
;               int variable in which the character count is
;               to be stored
;
NEWLINE  EQU     0ah           ;the linefeed character is C's
                               ; newline character
         DOSSEG
         .MODEL  small
         .CODE
         PUBLIC  _LineCount
_LineCount       PROC
         push    bp
         mov     bp,sp
         push    si            ;preserve calling program's
                               ; register variable, if any
         mov     si,[bp+4]     ;point SI to the string
         sub     cx,cx         ;set character count to 0
         mov     dx,cx         ;set line count to 0
LineCountLoop:
         lodsb                 ;get the next character
         and     al,al         ;is it null, to end the string?
         jz      EndLineCount  ;yes, we're done
         inc     cx            ;no, count another character
         cmp     al,NEWLINE    ;is it a newline?
         jnz     LineCountLoop ;no, check the next character
         inc     dx            ;yes, count another line
         jmp     LineCountLoop
EndLineCount:
         inc     dx            ;count the line that ends with the
                               ; null character
```

```
        mov     bx,[bp+6]       ;point to the location at which to
                                ; return the character count
        mov     [bx],cx         ;set the character count variable
        mov     ax,dx         ;return line count as function value
        pop     si            ;restore calling program's register
                              ; variable, if any
        pop     bp
        ret
_LineCount      ENDP
        END
```

The following C module, CALLCT.C, is a sample invocation of the *LineCount* function:

```
char * TestString="Line 1\nline 2\nline 3";
extern unsigned int LineCount(char * StringToCount,
      unsigned int * CharacterCountPtr);
main()
{
   unsigned int LCount;
   unsigned int CCount;

   LCount = LineCount(TestString, &CCount);
   printf("Lines: %d\nCharacters: %d\n", LCount, CCount);
}
```

The two modules are compiled and linked together with the command line

```
tcc -ms callct count.asm
```

As shown here, *LineCount* will only work when linked to small-model C programs, since pointer sizes and locations on the stack frame change in other models. Here's a version of *LineCount*, COUNTLG.ASM, that will work with large-model C programs (but not small-model ones, unless far pointers are passed, and *LineCount* is declared far):

```
; Large model C-callable assembler function to count the number
; of lines and characters in a zero-terminated string.
;
; Function prototype:
;       extern unsigned int LineCount(char * far StringToCount,
;               unsigned int * far CharacterCountPtr);
;       char far * StringToCount: pointer to the string on which
;                                 a line count is to be performed
;
;       unsigned int far * CharacterCountPtr: pointer to the
;               int variable in which the character count
;               is to be stored
```

```
;
NEWLINE  EQU     0ah        ;the linefeed character is C's newline
                            ; character
         .MODEL  large
         .CODE
         PUBLIC  _LineCount
_LineCount       PROC
         push    bp
         mov     bp,sp
         push    si                ;preserve calling program's
                                   ; register variable, if any
         push    ds                ;preserve C's standard data seg
         lds     si,[bp+6]         ;point DS:SI to the string
         sub     cx,cx             ;set character count to 0
         mov     dx,cx             ;set line count to 0
LineCountLoop:
         lodsb                     ;get the next character
         and      al,al            ;is it null, to end the string?
         jz       EndLineCount     ;yes, we're done
         inc      cx               ;no, count another character
         cmp      al,NEWLINE       ;is it a newline?
         jnz      LineCountLoop    ;no, check the next character
         inc      dx               ;yes, count another line
         jmp      LineCountLoop
EndLineCount:
         inc      dx               ;count line ending with null
                                   ; character
         les      bx,[bp+10]       ;point ES:BX to the location at
                                   ; which to return char count
         mov      es:[bx],cx       ;set the char count variable
         mov      ax,dx            ;return the line count as
                                   ; the function value
         pop      ds               ;restore C's standard data seg
         pop      si               ;restore calling program's
                                   ; register variable, if any
         pop      bp
         ret
_LineCount        ENDP
         END
```

COUNTLG.ASM can be linked to CALLCT.C with the following command line:

```
tcc -ml callct countlg.asm
```

Pascal calling conventions

See Chapter 8 for more information about Pascal calling conventions.

So far, you've seen how C normally passes parameters to functions by having the calling code push parameters right to left, call the function, and discard the parameters from the stack after the call. Turbo C is also capable of following the conventions used by Pascal programs in which parameters are passed from left to right and the *called* program discards the parameters from the stack. In Turbo C, Pascal conventions are enabled with the **-p** command-line option or the **pascal** keyword.

Here's an example of an assembler function that uses Pascal conventions:

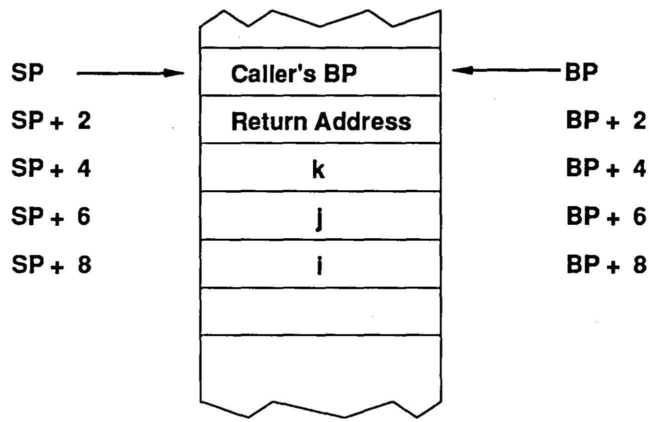
```
;
; Called as: TEST(i, j, k);
;
i       equ     8           ;leftmost parameter
j       equ     6
k       equ     4           ;rightmost parameter
;
        .MODEL  small
        .CODE
        PUBLIC  TEST
TEST    PROC
        push    bp
        mov     bp,sp
        mov     ax,[bp+i]   ;get i
        add     ax,[bp+j]   ;add j to i
        sub     ax,[bp+k]   ;subtract k from the sum
        pop     bp
        ret     6           ;return, discarding 6 parameter bytes
TEST    ENDP
        END
```

Figure 7.7 shows the stack frame after **MOV BP,SP** has been executed.

Note that **RET 6** is used by the called function to clear the passed parameters from the stack.

Pascal calling conventions also require all external and public symbols to be in uppercase, with no leading underscores. Why would you ever want to use Pascal calling conventions in a C program? Code that uses Pascal conventions tends to be somewhat smaller and faster than normal C code, since there's no need to execute an **ADD SP** *n* instruction to discard the parameters after each call.

Figure 7.7
State of the stack immediately after MOV BP, SP



Calling Turbo C from Turbo Assembler

Although it's most common to call assembler functions from C to perform specialized tasks, you may on occasion want to call C functions from assembler. As it turns out, it's actually easier to call a Turbo C function from a Turbo Assembler function than the reverse, since no stack-frame handling on the part of the assembler code is required. Let's take a quick look at the requirements for calling Turbo C functions from assembler.

Link in the C startup code

As a general rule, it's a good idea to only call Turbo C library functions from assembler code in programs that link in the C startup module as the first module linked. This "safe" class includes all programs that are linked from TC.EXE or with a TCC.EXE command line, and programs that are linked directly with TLINK that have C0T, C0S, C0C, C0M, C0L, or C0H as the first file to link.

You should generally not call Turbo C library functions from programs that don't link in the C startup module, since some Turbo C library functions will not operate properly if the startup code is not linked in. If you really want to call Turbo C library functions from such programs, we suggest you look at the startup source code (the file C0.ASM on the Turbo C distribution disks) and purchase the C library source code from Borland, so you can be sure to provide the proper initialization for the library functions you need. Another possible approach is to simply link each desired library function to an assembler program, called X.ASM for instance, which does nothing but call each function, linking them together with a command line like this:

```
tlink x,x,,cm.lib
```

where *m* is the first letter of the desired memory model (*t* for tiny, *s* for small, and so on). If TLINK reports any undefined symbols, then that library function can't be called unless the C startup code is linked into the program.

Note: Calling user-defined C functions that in turn call C library functions falls into the same category as calling library functions directly; lack of the C startup can potentially cause problems for *any* assembler program that calls C library functions, directly or indirectly.

Make sure you've got the right segment setup

As we learned earlier, you must make sure that Turbo C and Turbo Assembler are using the same memory model and that the segments you use in Turbo Assembler match those used by Turbo C. Refer to the previous section, "The framework," (page 282) if you need a refresher on matching memory models and segments. Also, remember to put **EXTRN** directives for far symbols either outside all segments or inside the correct segment.

Performing the call

You've already learned how Turbo C prepares for and executes function calls in the section "Calling Turbo Assembler functions from Turbo C" on page 281. We'll briefly review the mechanics of C function calls, this time from the perspective of calling Turbo C functions from Turbo Assembler.

All you need to do when passing parameters to a Turbo C function is push the rightmost parameter first, then the next rightmost parameter, and so on, until the leftmost parameter has been pushed. Then just call the function. For example, when programming in Turbo C, to call the Turbo C library function **strcpy** to copy *SourceString* to *DestString*, you would enter

```
strcpy(DestString, SourceString);
```

To perform the same call in assembler, you would use

```
lea  ax,SourceString   ;rightmost parameter
lea  bx,DestString     ;leftmost parameter
push ax                ;push rightmost first
push bx                ;push leftmost next
call _strcpy           ;copy the string
add  sp,4              ;discard the parameters
```

Don't forget to discard the parameters by adjusting SP after the call.

You can simplify your code and make it language independent at the same time by taking advantage of Turbo Assembler's **CALL** instruction extension:

```
call destination [language [,arg1] ...]
```

where *language* is C, PASCAL, BASIC, FORTRAN, PROLOG or NOLANGUAGE, and *arg* is any valid argument to the routine that can be directly pushed onto the processor stack.

Using this feature, the preceding code can be reduced to

```
lea   ax,SourceString
lea   bx,DestString
call  strcpy c,bx,ax
```

Turbo Assembler automatically inserts instructions to push the arguments in the correct order for C (AX first, then BX), performs the call to **_strcpy** (Turbo Assembler automatically inserts an underscore in front of the name for C), and cleans up the stack after the call.

If you're calling a C function that uses Pascal calling conventions, you have to push the parameters left to right and not adjust SP afterward:

```
lea   bx,DestString     ;leftmost parameter
lea   ax,SourceString   ;rightmost parameter
push  bx                ;push leftmost first
push  ax                ;push rightmost next
call  STRCPY            ;copy the string
                        ;leave the stack alone
```

Again, you can use Turbo Assembler's **CALL** instruction extension to simplify your code:

```
lea   bx,DestString      ;leftmost parameter
```

```
lea   ax,SourceString    ;rightmost parameter
call  strcpy pascal,bx,ax
```

Turbo Assembler automatically inserts instructions to push the arguments in the correct order for Pascal (BX first, then AX) and performs the call to **STRCPY** (converting the name to all uppercase, as is the Pascal convention).

Of course, the last example assumes that you've recompiled **strcpy** with the **-p** switch, since the standard library version of **strcpy** uses C rather than Pascal calling conventions. C functions return values as described in the section "Returning values" (page 302); 8- and 16-bit values in AX, 32-bit values in DX:AX, floating-point values in the 8087 TOS register, and structures in various ways according to size.

Rely on C functions to preserve the following registers and *only* the following registers: SI, DI, BP, DS, SS, SP, and CS. Registers AX, BX, CX, DX, ES, and the flags may be changed arbitrarily.

Calling a Turbo C function from Turbo Assembler

One case in which you might wish to call a Turbo C function from Turbo Assembler is when you need to perform complex calculations. This is especially true when mixed integer and floating-point calculations are involved; while it's certainly possible to perform such operations in assembler, it's simpler to let C handle the details of type conversion and floating-point arithmetic.

Let's look at an example of assembler code that calls a Turbo C function in order to get a floating-point calculation performed. In fact, let's look at an example in which a Turbo C function passes a series of integer numbers to a Turbo Assembler function, which sums the numbers and in turn calls another Turbo C function to perform the floating-point calculation of the average value of the series.

The C portion of the program in CALCAVG.C is

```
extern float Average(int far * ValuePtr, int NumberOfValues);
#define NUMBER_OF_TEST_VALUES 10
int TestValues[NUMBER_OF_TEST_VALUES] = {
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};

main()
{
```

```
   printf("The average value is: %f\n",
          Average(TestValues, NUMBER_OF_TEST_VALUES));
}
float IntDivide(int Dividend, int Divisor)
{
   return( (float) Dividend / (float) Divisor );
}
```

and the assembler portion of the program in AVERAGE.ASM is

```
;
; Turbo C-callable small-model function that returns the average
; of a set of integer values. Calls the Turbo C function
; IntDivide() to perform the final division.
;
; Function prototype:
;   extern float Average(int far * ValuePtr, int NumberOfValues);
;
; Input:
;   int far * ValuePtr:         ;the array of values to average
;   int NumberOfValues:         ;the number of values to average

        .MODEL  small
        EXTRN   _IntDivide:PROC
        .CODE
        PUBLIC  _Average
_Average        PROC
        push    bp
        mov     bp,sp
        les     bx,[bp+4]         ;point ES:BX to array of values
        mov     cx,[bp+8]         ;# of values to average
        mov     ax,0              ;clear the running total
AverageLoop:
        add     ax,es:[bx]        ;add the current value
        add     bx,2              ;point to the next value
        loop    AverageLoop
        push    WORD PTR [bp+8]   ;get back the number of values
                                  ; passed to IntDivide as the
                                  ; rightmost parameter
        push    ax      ;pass the total as the leftmost parameter
        call    _IntDivide  ;calculate the floating-point average
        add     sp,4              ;discard the parameters
        pop     bp
        ret                  ;average is in 8087's TOS register
_Average        ENDP
        END
```

The C **main** function passes a pointer to the array of integers *TestValues* and the length of the array to the assembler function *Average*. *Average* sums the integers, then passes the sum and the number of values to the C function *IntDivide*. *IntDivide* casts the sum and number of values to floating-point numbers and calculates the average value, doing in a single line of C code what would have taken several assembler lines. *IntDivide* returns the average to *Average* in the 8087 TOS register, and *Average* just leaves the average in the TOS register and returns to **main**.

CALCAVG.C and AVERAGE.ASM could be compiled and linked into the executable program CALCAVG.EXE with the command

```
tcc calcavg average.asm
```

Note that *Average* will handle both small and large data models without the need for any code change, since a far pointer is passed in all models. All that would be needed to support large code models (huge, large, and medium) would be use of the appropriate **.MODEL** directive.

Taking full advantage of Turbo Assembler's language-independent extensions, the assembly code in the previous example could be written more concisely as

```
        DOSSEG
        .MODEL   small,C
        EXTRN    C IntDivide:PROC
        .CODE
        PUBLIC   C Average
Average          PROC  C ValuePtr:DWORD,NumberOfValues:WORD
        les      bx,ValuePtr
        mov      cx,NumberOfValues
        mov      ax,0
AverageLoop:
        add      ax,es:[bx]
        add      bx,2                ;point to the next value
        loop     AverageLoop
        call     IntDivide C,ax,NumberOfValues
        ret
Average          ENDP
        END
```