Microsoft® Macro Assembler 5.1

Mixed-Language Programming Guide

Microsoft

Microsoft® Macro Assembler 5.1

Mixed-Language Programming Guide

Microsoft®

MIXED-LANGUAGE PROGRAMMING GUIDE

FOR THE MS-DOS® OPERATING SYSTEM

Document No. 410840031-500-R01-1287

TABLE OF CONTENTS

CONTENTS

CONTENTS

Figures

Tables

INTRODUCTION

Mixed-language programming is the process of creating programs from two or more source languages. This capability allows you to combine the unique strengths of Microsoft® BASIC, C, FORTRAN, Pascal, and Macro Assembler. Any one of these languages (in their recent versions) can call any of the others. Virtually all of the routines from all extensive language libraries are available to a mixed-language program.

For example, mixed-language programming helps you effectively use assembly language. You can develop the majority of your program quickly with Microsoft C or QuickBASIC, then call assembly for those few routines that are executed many times and must run with utmost speed.

Mixed-language programming also facilitates the transition from one language to another. You may have a large FORTRAN program which you are converting to C. You can replace your FORTRAN subroutines, one by one, with corresponding C functions. C-generated code can come on-line as soon as each function is written.

Finally, mixed-language programming is particularly valuable if you are marketing your own libraries. With the techniques presented here, you can produce libraries for any of the languages mentioned above, often with little change.

How to Use this Manual

This manual focuses on the concepts, syntax, and programming methods necessary to write mixed-language programs. The manual assumes that you have a basic understanding of the languages you wish to combine and that you already know how to write, compile, and link multiple-module programs with these languages. The manual does not attempt to teach the basics of programming in any particular language.

Mixed-language programming is not particularly difficult, but it does require that you understand certain basic issues. This manual first presents these issues in some detail. Once you understand the basics, you can proceed to the sections that are relevant to the languages you are using.

The manual is divided into two parts; each part has a different orientation and purpose:

- *Part 1. Mixed-Language Interfaces.*

  Part 1 shows how to establish an interface between any two languages (of those listed above). It does not assume you have any background in mixed-language programming, and extensively uses examples with simple parameter lists.

- *Part 2. Data Handling Reference.*

  Part 2 shows how to pass different kinds of data. This part of the manual assumes you already know the basics of mixed-language programming. It focuses on the particular programming considerations for passing strings, arrays, common blocks, etc.

Depending on your current level of knowledge, you may not need to start reading this manual at the beginning. The manual is structured so that you can easily turn to the section that would be most helpful:

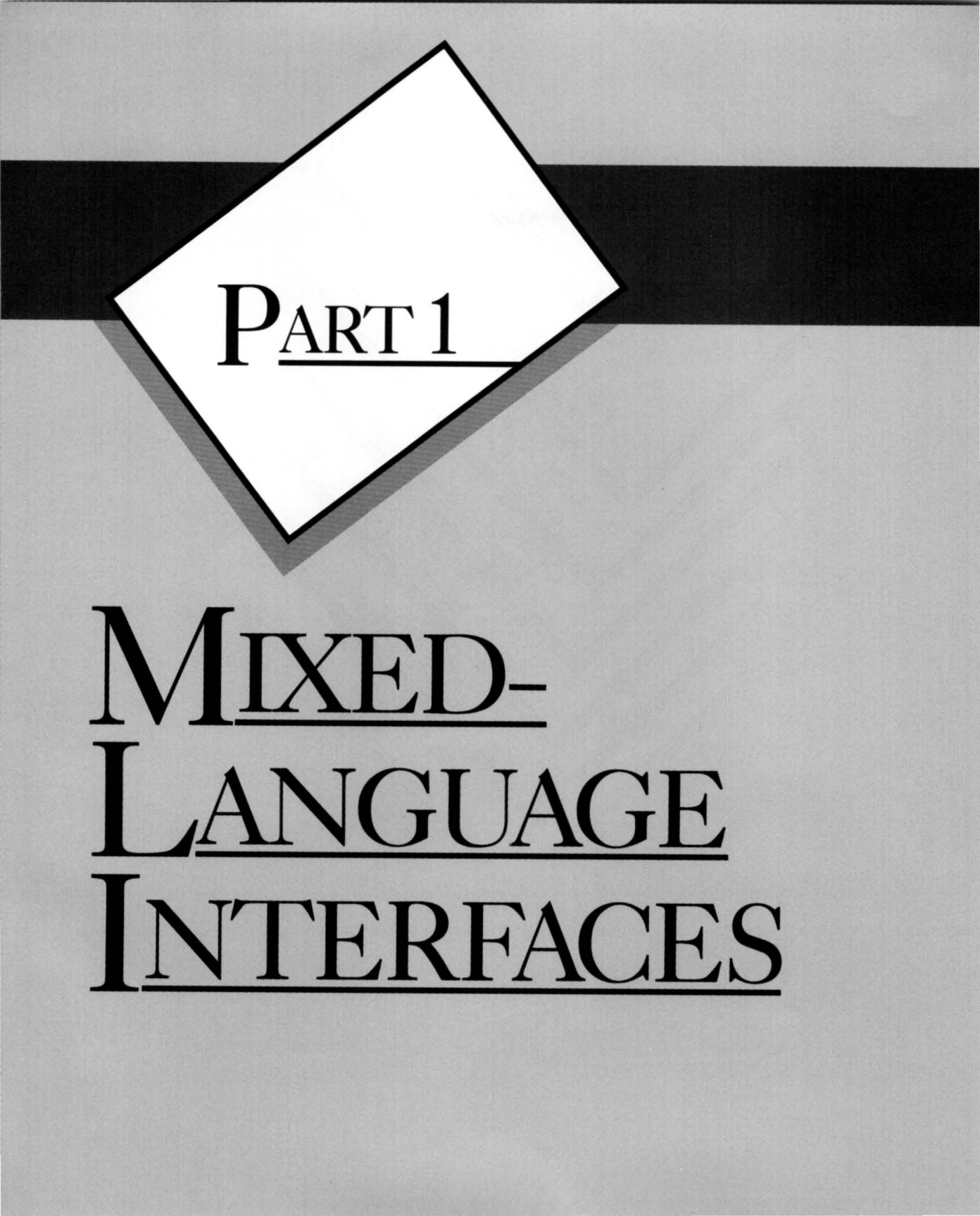
1. For an introduction to mixed-language concepts, read Chapter 1, "Elements of Mixed-Language Programming."
2. Depending on the high-level language of your main program, read either Chapter 2, 3, 4, or 5. The opening section of each of these chapters describes mixed-language syntax in detail. For quick reference, turn directly to the section in Chapters 2–5 most relevant to your combination of languages.
3. To find out about calls to assembly language, read Chapter 6, "Assembly-to-High-Level Interface."
4. After you have learned how to pass simple arguments (such as integers) between languages, use Part 2 as a reference for passing more complex kinds of data, such as strings and arrays.

Definitions

The notational conventions used in this manual are consistent with the conventions described in the user's guide for each Microsoft language. However, the following terms are used in specialized ways:

Term	Definition
Routine	Any function, subprogram, subroutine, or procedure that can be called from another language. The concept is similar to that of a procedure in assembly language; however, the term "routine" is used in most contexts to avoid confusion with the Pascal keyword **procedure**.

Parameter	A piece of data passed directly between two routines. (External data are shared by all routines, but cannot be said to be passed.) Although elsewhere the term "argument" is sometimes used interchangeably with "parameter," in this manual, "argument" is used to refer to the particular values or expressions given for parameters.
Interface	A method for providing effective communication between different formats. With high-level languages, an interface is often established by some kind of formal declaration.
Formal parameter	A formal parameter is a dummy parameter declared in an interface statement or declaration. C uses parameter type declarations rather than formal parameters.


PART 1
MIXED-LANGUAGE INTERFACES

CHAPTERS

PART 1 ◇ MIXED-LANGUAGE INTERFACES

Part 1 of the *Mixed-Language Programming Guide* explains how to establish an interface between modules written in Microsoft BASIC, C, FORTRAN, Pascal, and Macro Assembler. This part of the manual extensively uses examples; however, these examples feature only integer parameters, which are relatively easy to pass. Sharing other kinds of data (such as strings and arrays) presents special problems, and is dealt with in Part 2.

CHAPTER 1

ELEMENTS OF MIXED-LANGUAGE PROGRAMMING

Microsoft languages have special keywords that facilitate mixed-language programming (described in Chapters 2–5). However, in order to use these keywords, you first need to understand the basic issues involved.

This chapter describes the elements of mixed-language programming: how languages differ and how to resolve these differences. If you understand the principles described in the next few paragraphs, then you may want to turn directly to other chapters in this manual. Nevertheless, you still may find it helpful to refer to this chapter occasionally.

Section 1.1 presents the basic context of a mixed-language call, when and how you make such a call.

Sections 1.2–1.4 present the three fundamental mixed-language programming requirements:

- Naming convention requirement
- Calling convention requirement
- Parameter-passing requirement

Section 1.5 presents the compile-time and link-time issues, including use of memory models.

1.1 Making Mixed-Language Calls

Mixed-language programming always involves a call; specifically, it involves a function, procedure, or subroutine call. For example, a BASIC main module may need to execute a specific task that you would like to program separately. Instead of calling a BASIC subprogram, however, you decide to call a C function.

Mixed-language calls necessarily involve multiple modules (at least with Microsoft languages). Instead of compiling all of your source modules with the same compiler, you use different compilers. In the example mentioned above, you would compile the main-module source file with the BASIC compiler, another source file (written in C) with the C compiler, and then link together the two object files.

Figure 1.1 illustrates how the syntax of a mixed-language call works, using the example mentioned above.

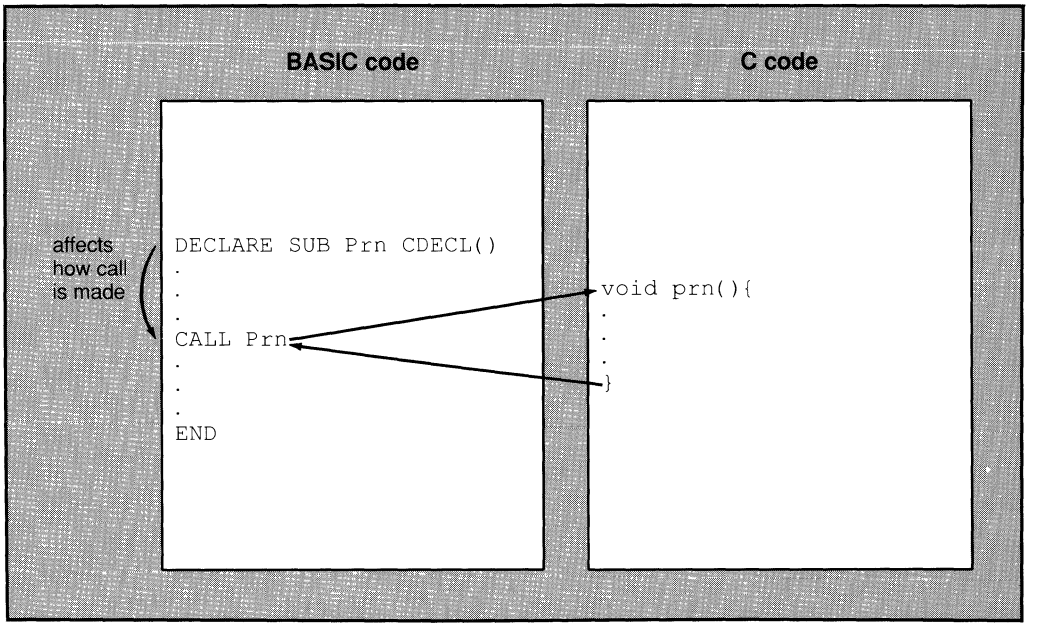


Figure 1.1 Mixed-Language Call

In the illustration above, the BASIC call to C is `CALL Prn`, similar to a call to a BASIC subprogram. However, there are two differences between this mixed-language call and a call between two BASIC modules: 1) the subprogram `Prn` is actually implemented in C, using standard C syntax; and 2) the implementation of the call in BASIC is affected by **DECLARE** statement, which uses the **CDECL** keyword in order to create compatibility with C. The **DECLARE** statement (which is discussed in detail in Chapter 2) is an example of a mixed-language "interface" statement. Each language provides its own form of interface.

Despite syntactic differences, functions, procedures, and FORTRAN subroutines are all similar. The principal difference is that some kinds of routines return values, and others do not. You can interchange routines that have a return value, and you can interchange routines that have no return value. (Note that in this manual, "routine" refers to any function, procedure or subroutine that can be called from another module.)

Table 1.1 shows the correspondence between routine calls in different languages.

Table 1.1
Language Equivalents for Routine Calls

Language	Return value	No return value
BASIC	**FUNCTION** procedure	subprogram
C	function	(void) function
FORTRAN	function	subroutine
Pascal	function	procedure
Macro Assembler	procedure	procedure

For example, a BASIC module can make a subprogram call to a FORTRAN subroutine. BASIC should make a **FUNCTION** call in order to call a FORTRAN function; otherwise, the call can be made, but the return value will be lost.

Note

BASIC **DEF FN** functions and **GOSUB** subroutines cannot be called from another language.

1.2 Naming Convention Requirement

The term "naming convention" refers to the way that a compiler alters the name of the routine before placing it into an object file.

It is important that you adopt a compatible naming convention when you issue a mixed-language call. If the name of the called routine is stored differently in each object file, then the linker will not be able to find a match. It will instead report an unresolved external.

Microsoft compilers place machine code into object files; but they also place there the names of all routines and variables which need to be

accessed publicly. That way, the linker can compare the name of a routine called in one module to the name of a routine defined in another module, and recognize a match. Names are stored in ASCII (American Standard Code for Information Interchange) format. You can see precisely how they are stored if you use the **DEBUG** utility to dump an object file's bytes.

BASIC, FORTRAN, and Pascal use roughly the same naming convention. They translate each letter to uppercase. BASIC type declaration characters (%, &, !, #, $) are dropped.

However, each language recognizes a different number of characters. FORTRAN recognizes the first 6 characters of any name, Pascal the first 8, and BASIC the first 40. If a name is longer than the language will recognize, additional characters are simply not placed in the object file.

C uses a quite different convention; the C compiler does not translate any letters to uppercase, but inserts a leading underscore (_) in front of the name of each routine. C recognizes the first 31 characters of a name.

Differences in naming conventions are taken care of for you automatically by mixed-language keywords, as long as you follow two rules:

1. If you are using any FORTRAN routines, all names should be 6 characters or less in length.
2. Do not use the **/NOIGNORE** linker option (which causes the linker to distinguish between `Prn` and `prn`). With C modules, this means that you will have to be careful not to rely upon differences between uppercase and lowercase letters.

   The **CL** driver and Microsoft QuickC™ automatically use the **/NOIGNORE** option when linking. To solve the problems created by this behavior, either link separately with the **LINK** utility, or use all lowercase letters in your C modules.

Figure 1.2 illustrates a complete mixed-language development example, showing how naming conventions enter into the process.

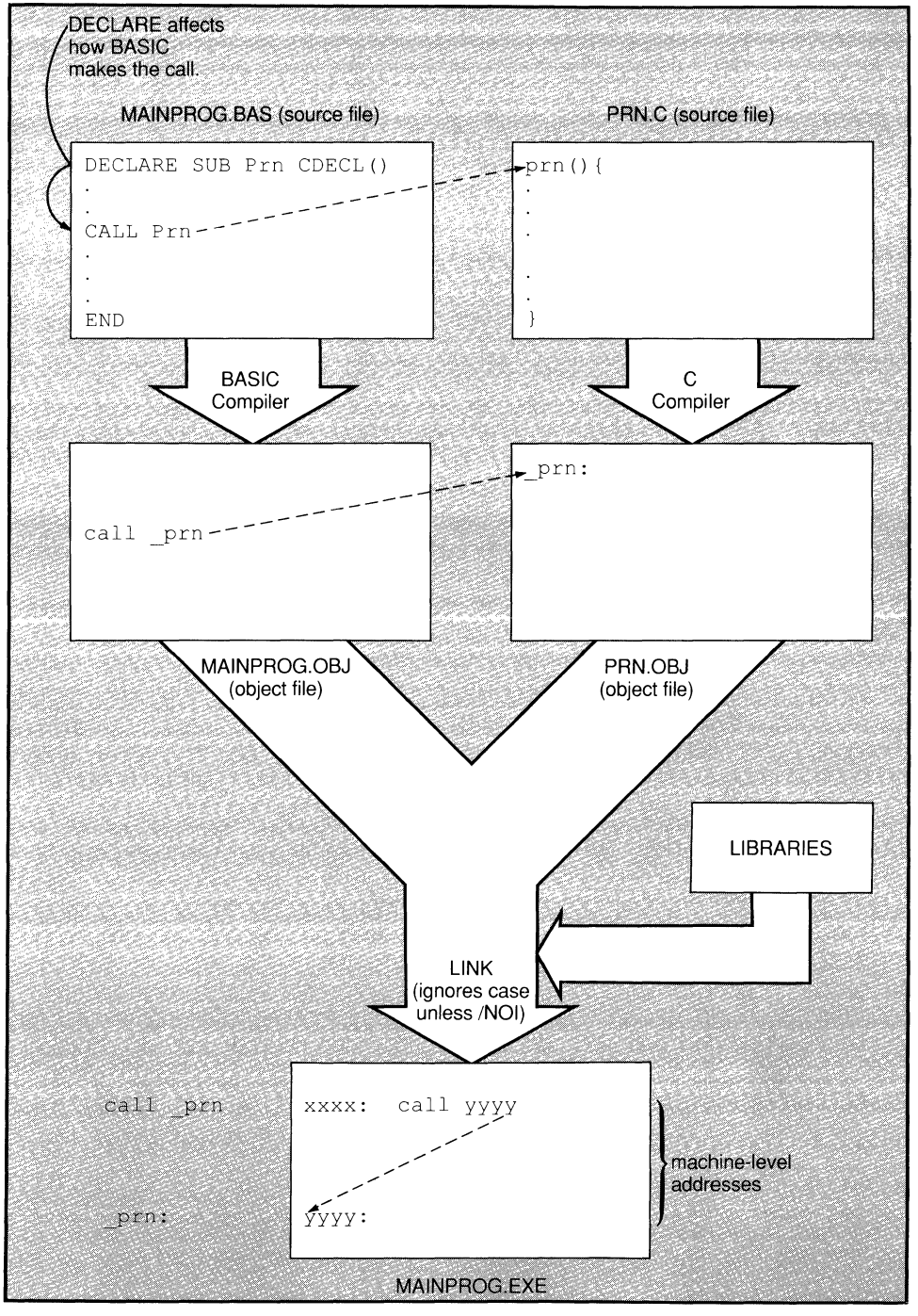


Figure 1.2 Naming Convention

In the example above, note that the BASIC compiler inserts a leading underscore in front of `Prn` as it places the name into the object file, because the **CDECL** keyword directs the BASIC compiler to use the C naming convention. BASIC will also convert all letters to lowercase when this keyword is used. (Strictly speaking, converting letters to lowercase is not part of the C naming convention; however, it is consistent with the programming style of most C programs.)

1.3 Calling Convention Requirement

The term "calling convention" refers to the way that a language implements a call. The choice of calling convention affects the actual machine instructions that a compiler generates in order to execute (and return from) a function, procedure, or subroutine call.

The calling convention is a low-level protocol. It is crucial that the two routines concerned (the routine issuing a call and the routine being called) recognize the same protocol. Otherwise, the processor may receive inconsistent instructions, thus causing the system to crash.

The use of a calling convention affects programming in two ways:

1. The calling routine uses a calling convention to determine in what order to *pass* arguments (parameters) to another routine. This convention can usually be specified in a mixed-language interface.
2. The called routine uses a calling convention to determine in what order to *receive* the parameters that were passed to it. In most languages, this convention can be specified in the routine's heading. BASIC, however, always uses its own convention to receive parameters.

In other words, each *call* to a routine uses a certain calling convention, and each routine *heading* specifies or assumes some calling convention. The two conventions must be compatible. With each language except BASIC, it is possible to change either calling convention. Usually, however, it is simplest to adopt the convention of the called routine. For example, a C function would use its own convention to call another C function, and use the Pascal convention to call Pascal.

BASIC, FORTRAN, and Pascal use the same standard calling convention. C, however, uses a quite different convention.

Note

The next few paragraphs discuss some of the details of calling conventions. It is not crucial for a high-level language programmer to understand these details; the programmer only needs to know that the different conventions are not compatible with each other.

The Microsoft BASIC, FORTRAN and Pascal calling conventions each push parameters onto the stack in the order in which they appear in the source code. For example, the BASIC statement `CALL Calc(A,B)` pushes the argument `A` onto the stack before it pushes `B`. These conventions also specify that the stack is restored by the called routine, just before returning control to the caller. (The stack is restored by removing parameters.)

The C calling convention pushes parameters onto the stack in the reverse order in which they appear in the source code. For example, the C function call `calc(a,b);` pushes `b` onto the stack before it pushes `a`. In contrast with the other high-level languages, the C calling convention specifies that a calling routine always restores the stack immediately after the called routine returns control.

The BASIC, FORTRAN, and Pascal conventions produce slightly less object code. However, the C convention makes calling with a variable number of parameters possible. (Because the first parameter is always the last one pushed, it is always on the top of the stack; therefore it has the same address relative to the frame pointer, regardless of how many parameters were actually passed.)

1.4 Parameter-Passing Requirement

Section 1.3 discussed the overall protocol (the calling convention) that two routines use to communicate with each other; this section concerns how an individual piece of data (a parameter) is actually sent.

If your routines do not agree on how a parameter is to be sent, then a called routine will receive bad data. It is also possible that the program could cause the system to crash.

Microsoft compilers support three methods for passing a parameter:

Method	Description
By near reference	Passes a variable's near (offset) address. This method gives the called routine direct access to the variable itself. Any change the routine makes to the parameter will be reflected in the calling routine.
By far reference	Passes a variable's far (segmented) address. This method is similar to passing by near reference, except that a longer address is passed. This method is slower than passing by near reference but is necessary when you pass data that is outside of the default data segment. (This is not an issue in BASIC or Pascal, unless you have specifically requested far memory.)
By value	Passes only the variable's value, not address. With this method, the called routine knows the value of the parameter, but has no access to the original variable. Changes to a value parameter have no affect on the value of the parameter in the calling routine, once the routine terminates.

The fact that there are different parameter-passing methods has two implications for mixed-language programming.

First, you need to make sure that the called routine and the calling routine use the same method for passing each parameter (argument). In most cases, you will need to check the parameter-passing defaults used by each language, and possibly make adjustments. Each language has keywords or language features that allow you to change parameter-passing methods.

Second, you may want to use a particular parameter-passing method rather than using the defaults of any language. (In fact, the examples in Chapters 2–5 specifically require one particular method or another, because of program logic.)

Table 1.2 summarizes the parameter-passing defaults for each language.

Table 1.2
Parameter-Passing Defaults

Language	Near reference	Far reference	By value
BASIC	all		
C	near arrays	far arrays	non-arrays
FORTRAN		all	all[1]
Pascal	**VAR, CONST**	**VARS, CONSTS**	other params

[1] When a PASCAL or C attribute is applied to a FORTRAN routine, pass by value becomes the default.

Each language has methods for overriding these defaults, which are listed in Chapter 7.

1.5 Compiling and Linking

After you have written your source files and resolved the issues raised in Sections 1.2–1.4, you are ready to compile individual modules and then link them together.

1.5.1 Compiling with Proper Memory Models

With Microsoft BASIC, FORTRAN, and Pascal, no special options are required to compile source files that are part of a mixed-language program.

With Microsoft C, however, you need to be aware that not all memory models will be compatible with other languages. BASIC, FORTRAN, and Pascal use only far (segmented) code addresses. Therefore, you must always compile C modules in medium, large, or huge model, because these models also use far code addresses. Compiling in small or compact model will cause the mixed-language program to crash, as soon as a call is made to or from C. (This problem can be averted if you apply the **far** keyword to a C function definition, in order to specify that the function uses a far call and return.)

The paragraph above concerns the size of code addresses. Differences in the size of data addresses can be resolved through compile options or in the source code. Choice of memory model affects the default data pointer size in C and FORTRAN, although this default can be overriden with **near** and **far**. Choice of memory model, with C and FORTRAN, also affects whether data objects are located in the default data segment; if a data object is not located in the default data segment, it cannot be directly passed by near reference.

1.5.2 Linking with Language Libraries

In many cases, linking modules compiled with different languages can be done easily. Any of the following measures will ensure that all of the required libraries are linked in the correct order:

- Put all language libraries in the same directory as the source files.
- List directories containing all needed libraries in the **LIB** environment variable.
- Let the linker prompt you for libraries.

In each of the above cases, the linker finds libraries in the order that it requires them. If you enter the libraries on the command line, then they must be entered in a particular order.

However, if you are using Version 4.0 of FORTRAN to produce one of your modules, then you need to link with **/NOD** (no default libraries), and specify all the libraries you need directly on the link command line. You can also specify these libraries with an automatic-response file (or batch file), but you cannot use a default-library search.

If your program uses both FORTRAN and C, then specify the library for the most recent of the two language products first. If you use C 4.0 with FORTRAN 4.0 or later, specify the FORTRAN library first. If you use C 5.0 or later, specify the C library first. In addition, make sure that you chose a C-compatible library when you installed FORTRAN.

If you are listing BASIC libraries on the **LINK** command line, the BASIC libraries must precede all others.

■ Example

```
LINK /NOD mod1 mod2,,,GRAFX+LLIBCE+LLIBFORE
```

The example above links two object modules with the C 5.1 and FORTRAN 4.0 large-model libraries. In addition, the example links in an extra library, GRAFX.

CHAPTER 2

BASIC CALLS TO HIGH-LEVEL LANGUAGES

Microsoft BASIC supports calls to routines written in Microsoft C, FORTRAN, and Pascal. This chapter describes the necessary syntax for calling these other languages, and then gives examples for each combination of BASIC with another language. Only integers are used as parameters in these examples.

The chapter ends with a section that lists restrictions on the use of functions from the C standard library. Consult this section if you are using any memory allocation or system library functions.

For information on how to pass specific kinds of data, consult Part 2, "Data Handling Reference."

2.1 The BASIC Interface to Other Languages

The BASIC **DECLARE** statement provides a flexible and convenient interface to other languages. It is available with Microsoft QuickBASIC, Versions 4.0 and later. Versions that do not provide the **DECLARE** statement do not provide libraries that are compatible with other languages, either. These earlier versions have limited use in mixed-language programs, as they cannot successfully call a C, FORTRAN, or Pascal routine that makes any use of a library.

The syntax of the **DECLARE** statement is summarized below.

2.1.1 The DECLARE Statement

When you call a function, the **DECLARE** statement syntax is as follows:

DECLARE FUNCTION *name* [[**CDECL**]][[**ALIAS** "*aliasname*"]][[(*parameter-list*)]]

When you call a subprogram, the statement syntax is as follows:

DECLARE SUB *name* [[**CDECL**]][[**ALIAS** "*aliasname*"]][[(*parameter-list*)]]

The *name* field is the name of the function or subprogram procedure you wish to call, as it appears in the BASIC source file. Here are the recommended steps for using the **DECLARE** statement to call other languages:

1. For each distinct interlanguage routine you plan to call, put a **DECLARE** statement in the BASIC source file before the routine is called.

   For example, your program may call the subprogram `Maxparam` five different times, each time with different arguments. However, you need to declare `Maxparam` just once. Ideally, **DECLARE** statements should be placed near the beginning of the source file.

2. If you are calling a routine from a C module, use **CDECL** in the **DECLARE** statement (unless the C routine is declared with the **pascal** or **fortran** keyword).

   CDECL directs BASIC to use the C naming and calling conventions during each subsequent call to *name*. No similar keywords are provided for Pascal or FORTRAN, because they each use the same calling convention as BASIC.

3. If you are calling a FORTRAN routine with a name longer than six characters, or a C or Pascal routine with a name longer than eight characters, use the **ALIAS** feature. The use of **ALIAS** is explained in the section below.

4. Use the parameter list to determine how each parameter is to be passed. The use of the parameter list is explained below, in the section immediately after the information on **ALIAS**.

5. Once the routine is properly declared, call it just as you would a BASIC subprogram or function.

The other fields are explained in the following discussion.

2.1.2 Using ALIAS

As noted above, the use of **ALIAS** may be necessary because FORTRAN places only the first 6 characters of a name into an object file, whereas C and Pascal each place the first 8, but BASIC will place up to 40 characters of a name into an object file.

Note

You do *not* need the **ALIAS** feature to remove type declaration characters (%, &, !, #, $). BASIC automatically removes these characters when it generates object code. Thus, `Fact%` in BASIC matches `Fact` in Pascal.

The **ALIAS** keyword directs BASIC to place *aliasname* into the object file, instead of *name*. The BASIC source file still contains calls to *name*. However, these calls are interpreted as if they were actually calls to *aliasname*.

■ Example

```
DECLARE FUNCTION Quadratic% ALIAS "QUADRA" (a, b, c)
```

In the example above, QUADRA, the *aliasname*, contains the first six characters of Quadratic%, the *name*. This causes BASIC to place QUADRA into the object code, thereby mimicking FORTRAN's behavior.

2.1.3 Using the Parameter List

The parameter list syntax is displayed below, followed by explanations of each field. Note that you can use **BYVAL** or **SEG**, but not both.

■ Syntax

[[**BYVAL** | **SEG**]] *variable* [[**AS** *type*]]...,

Use the **BYVAL** keyword to declare a value parameter. In each subsequent call, the corresponding argument will be passed by value (the default method for C and Pascal modules).

Note

BASIC provides two ways of "passing by value." The usual method of passing by value is to use an extra set of parentheses, as in:

```
CALL Holm((A))
```

This method actually creates a temporary value, whose address is passed. **BYVAL** provides a true method of passing by value, because the value itself is passed, not an address. Only by using **BYVAL** will a BASIC program be compatible with a non-BASIC routine that expects a value parameter.

Use the **SEG** keyword to declare a far reference parameter. In each subsequent call, the far (segmented) address of the corresponding argument will be passed (the default method for FORTRAN modules).

You can choose any legal name for *variable*; but only the type associated with the name has any significance to BASIC. As with other variables, the type can be indicated with a type declaration character (%, &, !, #, $) or by implicit declaration.

You can use the **AS** *type* clause to override the type declaration of *variable*. The *type* field can be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING**, a user-defined type, or **ANY**, which directs BASIC to permit any type of data to be passed as the argument.

■ **Examples**

```
DECLARE FUNCTION Calc2! CDECL (BYVAL a%, BYVAL b%, BYVAL c!)
```

In the example above, `Calc2` is declared as a C routine that takes three arguments: the first two are integers passed by value, and the last is a single-precision real number passed by value.

```
DECLARE SUB Maxout (SEG var1 AS INTEGER, BYVAL var2 AS DOUBLE)
```

This example declares a subprogram `Maxout` that takes an integer passed by far reference, and a double-precision real number passed by value.

2.2 Alternative BASIC Interfaces

Though the **DECLARE** statement provides a particularly convenient interface, there are other methods of implementing mixed-language calls.

Instead of modifying the behavior of BASIC with **CDECL**, you can modify the behavior of C by applying the **pascal** or **fortran** keyword to the function definition heading. (These two keywords are functionally equivalent). Or you can compile the C module with the **/Gc** option, which specifies that all C functions, calls, and public symbols use the conventions of BASIC/FORTRAN/Pascal.

For example, the following C function uses the BASIC/FORTRAN/Pascal conventions to receive an integer paramter:

```
int pascal fun1(n)
int n;
```

You can specify parameter-passing methods without using a **DECLARE** statement or by using a **DECLARE** statement and omitting the parameter list.

- You can make the call with the **CALLS** statement. The **CALLS** statement causes each parameter to be passed by far reference.
- You can use the **BYVAL** and **SEG** keywords in the actual parameter list when you make the call:

  ```
  CALL Fun2(BYVAL Term1, BYVAL Term2, SEG Sum);
  ```

In the example above, **BYVAL** and **SEG** have the same meaning that they have in a BASIC **DECLARE** statement. When you use **BYVAL** and **SEG** this way, however, you need to be careful because neither the type nor the number of parameters will be checked as they would be in a **DECLARE** statement.

2.3 BASIC Calls to C

This section applies the steps outlined in Section 2.1 to two example programs. An analysis of programming considerations follows each example.

2.3.1 Calling C from BASIC—No Return Value

The example below demonstrates a BASIC main module calling a C function, `maxparam`. The function `maxparam` returns no value, but adjusts the lower of two arguments to equal the higher argument.

■ Example

```
' BASIC source file - calls C function returning no value
'
DECLARE SUB Maxparam CDECL (A AS INTEGER, B AS INTEGER)
'
' DECLARE as subprogram, since there is no return value
' CDECL keyword causes Maxparam call to be made w/ C conventions
' Integer parameters passed by near reference (BASIC default)
'
X% = 5
Y% = 7
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% before call
CALL Maxparam(X%, Y%)                     ' Call C function
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% after call
END
```

```
/* C source file */
/* Compile in MEDIUM or LARGE memory model */
/* Maxparam declared VOID because no return value */

void maxparam(p1, p2)
int near *p1;         /* Integer params received by near ref. */
int near *p2;         /* NEAR keyword not needed in MEDIUM model */
{
        if (*p1 > *p2)
                *p2 = *p1;
        else
                *p1 = *p2;
}
```

Naming conventions: The **CDECL** keyword causes `Maxparam` to be called with the C naming convention (as `_maxparam`). Note that word length is not an issue because `maxparam` does not exceed eight characters.

Calling conventions: The **CDECL** keyword causes `Maxparam` to be called with the C calling convention, which pushes parameters in the reverse order to how they appear in the source code.

Parameter-passing methods: Since the C function `maxparam` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is the default for BASIC (so neither **BYVAL** nor **SEG** is used) and is specified in C by using near pointers.

Far reference could have been specified by applying **SEG** to each argument in the **DECLARE** statement. In that case, the C parameter declarations would use far pointers.

2.3.2 Calling C from BASIC—Function Call

The example below demonstrates a BASIC main module calling a C function, `fact`. This function returns the factorial of an integer value.

■ Example

```
' BASIC source file - calls C function with return value
'
DECLARE FUNCTION Fact% CDECL (BYVAL N AS INTEGER)
'
' DECLARE as function returning integer (%)
' CDECL keyword causes Fact% call to be made w/ C conventions
' Integer parameter passed by value
'
X% = 3
Y% = 4
PRINT USING "The factorial of X%    is ####"; Fact%(X%)
PRINT USING "The factorial of Y%    is ####"; Fact%(Y%)
PRINT USING "The factorial of X%+Y% is ####"; Fact%(X%+Y%)
END
```

```
/* C source file */
/* Compile in MEDIUM or LARGE model */
/* Factorial function, returning integer */

int fact(n)
int n;          /* Integer passed by value, the C default */
{
        int     result = 1;

        while (n > 0)
                result *= n--;  /* Parameter n modified here */
        return(result);
}
```

Naming conventions: The **CDECL** keyword causes `Fact` to be called with the C naming convention (as `_fact`). Note that word length is not an issue because `fact` does not exceed eight characters.

Calling conventions: The **CDECL** keyword causes `fact` to be called with the C calling convention, which pushes parameters in reverse order and specifies other low-level differences.

Parameter-passing methods: The C function above should receive the parameter by value. Otherwise the function will corrupt the parameter's value in the calling module. True passing by value is achieved in BASIC only by applying **BYVAL** to the parameter in the **DECLARE** statement; in C, passing by value is the default (except for arrays).

2.4 BASIC Calls to FORTRAN

This section applies the steps outlined in Section 2.1 to two example programs. An analysis of programming considerations follows each example.

2.4.1 Calling FORTRAN from BASIC—Subroutine Call

The example below demonstrates a BASIC main module calling a FORTRAN subroutine, MAXPARAM. The subroutine returns no value, but adjusts the lower of two arguments to equal the higher argument.

■ Example

```
' BASIC source file - calls FORTRAN subroutine
'
DECLARE SUB Maxparam ALIAS "MAXPAR" (A AS INTEGER, B AS INTEGER)
'
' DECLARE as subprogram, since there is no return value
' ALIAS needed because FORTRAN recognizes only first 6 letters
' Integer parameters passed by near reference (BASIC default)
'
X% = 5
Y% = 7
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% before call
CALL Maxparam(X%, Y%)                     ' Call FORTRAN function
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% after call
END
```

```
C   FORTRAN source file, subroutine MAXPARAM
C
        SUBROUTINE MAXPARAM (I, J)
        INTEGER*2 I [NEAR]
        INTEGER*2 J [NEAR]
C
C   I and J received by near reference, because of NEAR attribute
C
        IF (I .GT. J) THEN
                J = I
        ELSE
                I = J
        ENDIF
        END
```

Naming conventions: By default, BASIC places all eight characters of Maxparam into the object file, yet FORTRAN places only the first six. This conflict is resolved with the **ALIAS** feature: both modules place MAXPAR into the object file.

Calling conventions: BASIC and FORTRAN use the same convention for calling.

Parameter-passing methods: Since the subprogram `Maxparam` may alter the value of either parameter, both arguments must be passed by reference. In this case, near reference was chosen; this method is the default for BASIC (so neither **BYVAL** nor **SEG** is used) and is specified in FORTRAN by applying the **NEAR** attribute to each of the parameter declarations.

Far reference could have been specified by applying **SEG** to each argument in the **DECLARE** statement. In that case, the **NEAR** attribute would not be used in the FORTRAN code.

2.4.2 Calling FORTRAN from BASIC— Function Call

The example below demonstrates a BASIC main module calling a FORTRAN function, `FACT`. This function returns the factorial of an integer value.

■ Example

```
' BASIC source file - calls FORTRAN function
'
DECLARE FUNCTION Fact% (BYVAL N AS INTEGER)
'
' DECLARE as function returning integer(%)
' Integer parameter passed by value
'
X% = 3
Y% = 4
PRINT USING "The factorial of X%    is ####"; Fact%(X%)
PRINT USING "The factorial of Y%    is ####"; Fact%(Y%)
PRINT USING "The factorial of X%+Y% is ####"; Fact%(X%+Y%)
END
```

```
C   FORTRAN source file - factorial function
C
        INTEGER*2 FUNCTION FACT (N)
        INTEGER*2 N [VALUE]
C
C   N is received by value, because of VALUE attribute
C
        INTEGER*2 I
        FACT = 1
        DO 100 I = 1, N
            FACT = FACT * I
  100       CONTINUE
        RETURN
        END
```

Naming conventions: There are no conflicts with naming conventions because the function name, FACT, does not exceed six characters. The type declaration character (%) is automatically dropped by BASIC.

Calling conventions: BASIC and FORTRAN use the same convention for calling.

Parameter-passing methods: When a parameter is passed that should not be changed, it is generally safest to pass the parameter by value. True passing by value is specified in BASIC by applying **BYVAL** to an argument in the **DECLARE** statement; in FORTRAN, passing by value is achieved by applying the **VALUE** attribute to a parameter declaration.

2.5 BASIC Calls to Pascal

This section applies the steps outlined in Section 2.1 to two example programs. An analysis of programming considerations follows each example.

2.5.1 Calling Pascal from BASIC— Procedure Call

The example below demonstrates a BASIC main module calling a Pascal procedure, Maxparam. Maxparam returns no value, but adjusts the lower of two arguments to equal the higher argument.

■ Example

```
' BASIC source file - calls Pascal procedure
'
DECLARE SUB Maxparam (A AS INTEGER, B AS INTEGER)
'
' DECLARE as subprogram, since there is no return value
' Integer parameters passed by near reference (BASIC default)
'
X% = 5
Y% = 7
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% before call
CALL Maxparam(X%, Y%)                     ' Call Pascal function
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% after call
END
```

```
{  Pascal source code - Maxparam procedure. }

module Psub;
 procedure Maxparam(var a:integer; var b:integer);

{  Two integer parameters are received by near reference. }
{  Near reference is specified with the VAR keyword. }

 begin
    if a > b then
        b := a
    else
        a := b
 end;
end.
```

Naming conventions: Note that word length is not an issue because Maxparam does not exceed eight characters.

Calling conventions: BASIC and Pascal use the same calling convention.

Parameter-passing methods: Since the procedure `Maxparam` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is the default for BASIC (so neither **BYVAL** nor **SEG** is used) and is specified in Pascal by declaring parameters as **VAR**.

Far reference could have been specified by applying **SEG** to each argument in the **DECLARE** statement. In that case, the **VARS** keyword would be required instead of **VAR**.

2.5.2 Calling Pascal from BASIC—Function Call

The example below demonstrates a BASIC main module calling a Pascal function, `Fact`. This function returns the factorial of an integer value.

■ Example

```
' BASIC source file - calls Pascal function
'
DECLARE FUNCTION Fact% (BYVAL N AS INTEGER)
'
' DECLARE as function returning integer (%)
' Integer parameter passed by value
'
X% = 3
Y% = 4
PRINT USING "The factorial of X%    is ####"; Fact%(X%)
PRINT USING "The factorial of Y%    is ####"; Fact%(Y%)
PRINT USING "The factorial of X%+Y% is ####"; Fact%(X%+Y%)
END
```

```
{ Pascal source code - factorial function. }

module Pfun;
 function Fact (n : integer) : integer;

{ Integer parameters received by value, the Pascal default. }

 begin
    Fact := 1;
    while n > 0 do
        begin
            Fact := Fact * n;
            n := n - 1;           { Parameter n altered here. }
        end;
 end;
end.
```

Naming conventions: Note that word length is not an issue because `fact` does not exceed eight characters.

Calling conventions: BASIC and Pascal use the same calling convention.

Parameter-passing methods: The Pascal function above should receive a parameter by value. Otherwise the function will corrupt the parameter's value in the calling module. True passing by value is achieved in BASIC only by applying **BYVAL** to the parameter; in Pascal, passing by value is the default.

2.6 Restrictions on Calls from BASIC

BASIC has a much more complex environment and initialization procedure than C, FORTRAN, or Pascal (all of which use a similar environment). Interlanguage calling between BASIC and these other languages is possible only because BASIC intercepts a number of library function calls from the other languages and handles them in its own way. In other words, BASIC creates a host environment in which the C, FORTRAN, and Pascal library routines can function.

However, BASIC is limited in its ability to handle some C function calls. This section considers two kinds of limitations: C memory-allocation functions, which may require a special declaration, and C library functions, which cannot be called at all.

2.6.1 Memory Allocation

If your C module is medium model and you do dynamic memory allocation with **malloc()**, or if you execute explicit calls to **_nmalloc()** with any memory model, then you need to include the following lines in your BASIC source code before you call C:

```
DIM mallocbuf%(2048)
COMMON SHARED /NMALLOC/ mallocbuf%()
```

The array can have any name; only the size of the array is significant. However, the name of the common block must be **NMALLOC**. In the QuickBASIC, Version 4.0, in-memory environment, you need to put this declaration in a module that you load into a resident Quick library.

The example above has the effect of reserving 4k bytes of space in the common block **NMALLOC**. When BASIC intercepts C **malloc** calls, BASIC allocates space out of this common block.

Warning

When you call the BASIC intrinsic function **CLEAR**, all space allocated with near **malloc** calls will be lost. If you use **CLEAR** at all, use it only before any calls to **malloc**.

When you make far memory requests in mixed-language programs, you may find it useful to first call the BASIC intrinsic function **SETMEM**. This function can be used to reduce the amount of memory BASIC is using, thus freeing up memory for far allocations.

2.6.2 Incompatible Functions

The following C functions are incompatible with BASIC and should be avoided:

- All forms of **spawn()** and **exec()**
- **system()**
- **getenv()**
- **putenv()**

In addition, you should not link with the *x***VARSTK.OBJ** modules (where *x* is a memory model), which C provides to allocate memory from the stack.

CHAPTER 3

C CALLS TO HIGH-LEVEL LANGUAGES

Microsoft C supports calls to routines written in Microsoft FORTRAN and Pascal. Also, if the main program is in BASIC, then a C routine can call a BASIC routine. This chapter describes the necessary syntax for calling these other languages, and then gives examples for each language. Only simple parameter lists are used in this chapter.

For information on how to pass particular kinds of data, consult Part 2, "Data Handling Reference."

3.1 The C Interface to Other Languages

The C interface to other languages utilizes the standard C **extern** statement, combined with the special **fortran** or **pascal** keyword. Using either of these keywords causes the routine to be called with the naming and calling conventions of FORTRAN/Pascal/BASIC. Here are the recommended steps for using this statement to execute a mixed-language call from C:

1. Write an **extern** statement for each mixed-language routine called.

   The **extern** statement should precede all calls to the routine. The exact rules of syntax for using the **fortran** and **pascal** keywords with the **extern** statement are presented below.

   Instead of using the **fortran** or **pascal** keyword, you can simply compile with **/Gc**. The **/Gc** option causes all functions in the module to use the BASIC/FORTRAN/Pascal naming and calling conventions, except where you apply the **cdecl** keyword.

2. Use parameter type declarations within the **extern** statement.

   This step is essential if you are going to specify pointers that are not the default size. (Near pointers are the default for medium model; far pointers are the default for large model.)

3. To pass an argument by reference, pass a pointer to the object.

   C automatically translates array names into addresses. Therefore, arrays are automatically passed by reference.

4. Once a routine has been properly declared with an **extern** statement, call it just as you would call a C function.

5. Always compile the C module in medium or large model.

■ Using fortran and pascal Keywords

There are two rules of syntax that apply when you use the **fortran** or **pascal** keyword:

1. The **fortran** and **pascal** keywords modify the item immediately to the right.
2. The special **near** and **far** keywords can be used with the **fortran** and **pascal** keywords in declarations. The sequences **fortran far** and **far fortran** are equivalent.

The keywords **pascal** and **fortran** actually have the same effect on the program; the use of one or the other makes no difference except for documentation purposes. Use either keyword to declare a BASIC routine.

The following examples demonstrate the syntax rules presented above.

■ Examples

```
extern short pascal thing(short, short);      /* Example 1 */
```

Example 1 declares `thing` to be a BASIC, Pascal, or FORTRAN function taking two **short** parameters and returning a **short** value.

```
extern void (fortran *thing)(long);           /* Example 2 */
```

Example 2 declares `thing` to be pointer to a BASIC, Pascal, or FORTRAN routine that takes a **long** parameter and returns no value. The keyword **void** is appropriate when the called routine is a BASIC subprogram, Pascal procedure, or FORTRAN subroutine, since it indicates that no return value is required.

```
extern short near pascal thing(double *);      /* Example 3 */
```

Example 3 declares `thing` to be a **near** BASIC, Pascal, or FORTRAN routine. The routine receives a **double** parameter by reference (because it expects a pointer to a **double**) and returns a **short** value.

```
extern short pascal near thing(double *);      /* Example 4 */
```

Example 4 is equivalent to Example 3 (`pascal near` is equivalent to `near pascal`).

3.2 Alternative C Interfaces

When you call BASIC, you must use the BASIC/FORTRAN/Pascal conventions to make the call. When you call FORTRAN or Pascal, however, you have a choice. You can make C adopt the appropriate conventions (as described in the previous section), or you can make the FORTRAN or Pascal routine adopt the C conventions.

To make a FORTRAN or Pascal routine adopt the C conventions, simply put the **C** attribute in the heading of the routine's definition. The following example demonstrates the syntax for the **C** attribute in a FORTRAN subroutine-definition heading:

```
SUBROUTINE FFROMC [C] (N)
INTEGER*2 N
```

The following example demonstrates the syntax for the **C** attribute in a Pascal procedure-definition heading:

```
module Pmod;
procedure Pfromc (n : integer) [C];
begin
```

3.3 C Calls to BASIC

No BASIC routine can be executed unless the main program is in BASIC, because a BASIC routine requires the environment to be initialized in a way that is unique to BASIC. No other language will perform this special initialization.

However, it is possible for a program to start up in BASIC, call a C function that does most of the work of the program, and then call BASIC subprograms and function procedures as needed. Figure 3.1 illustrates how this can be done.

The following rules are recommended when you call BASIC from C:

1. Start up in a BASIC main module. You will need to use QuickBASIC, Version 4.0 or later, and the **DECLARE** statement to

provide an interface to the C module. (See Chapter 2, "BASIC Calls to High-Level Languages," for more information.)

2. Once in the C module, declare the BASIC routine as **extern**, and include type information for parameters. Use either the **fortran** or **pascal** keyword to modify the routine itself.

3. Make sure that all data are passed as a near pointer. BASIC can *pass* data in a variety of ways, but is unable to receive data in any form other than near reference.

   With near pointers, the program assumes that the data are in the default data segment. If you want to pass data that are *not* in the default data segment (this is only a consideration with large-model programs), then first copy the data to a variable that is in the default data segment.

4. Compile the C module in medium or large model.

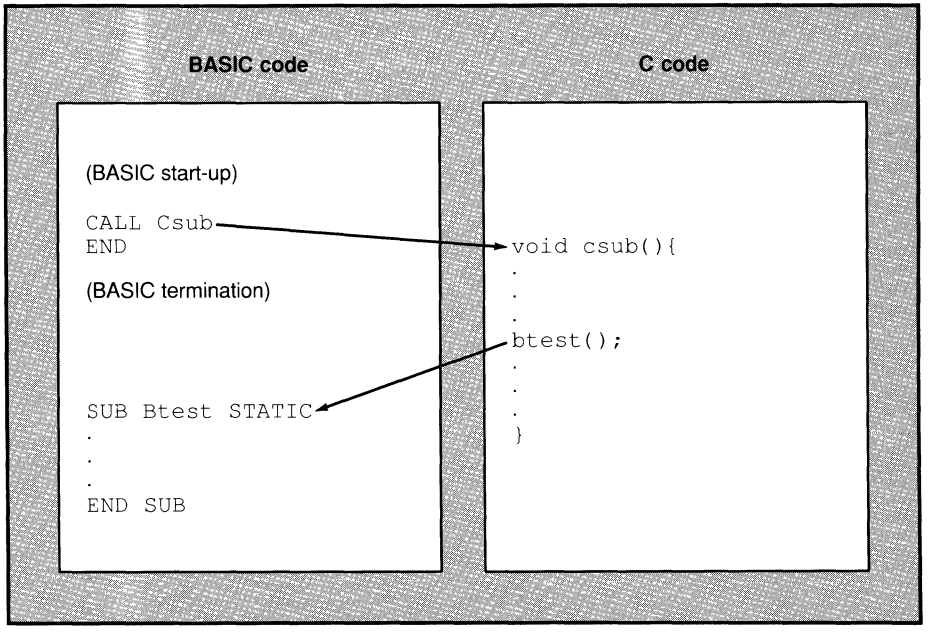


Figure 3.1 C Call to BASIC

The example below demonstrates a BASIC program that calls a C function. The C function then calls a BASIC function that returns twice the number passed it and a BASIC subprogram that prints two numbers.

■ Example

```
' BASIC source
'
DEFINT A-Z
DECLARE SUB Cprog CDECL()
CALL Cprog
END
'
FUNCTION Dbl(N) STATIC
    Dbl = N*2
END FUNCTION
'
SUB Printnum(A,B) STATIC
    PRINT "The first number is ";A
    PRINT "The second number is ";B
END SUB
```

```
/* C source; compile in medium or large model */

extern int fortran dbl(int near *);
extern void fortran printnum(int near *, int near *);

void cprog()
{
        int near a = 5;    /* NEAR guarantees that the data   */
        int near b = 6;    /*   will be placed in default     */
                           /*   data segment (DGROUP)         */

        printf("Twice of 5 is %d\n", dbl(&a));
        printnum(&a, &b);
}
```

In the example above, note that the *addresses* of `a` and `b` are passed, since BASIC expects to receive addresses for parameters. Also note that the keyword **near** is used to declare each pointer; this keyword would be unnecessary if we knew the C module was compiled in medium model rather than large.

Calling and naming conventions are resolved by the **CDECL** keyword in BASIC's declaration of `Cprog`, and by **fortran** in C's declaration of `dbl` and `printnum`.

Note

QuickBASIC 4.0 provides a number of "user-entry points," which are BASIC system-level functions that may be called directly from C. These functions provide memory management, C-string services, and I/O procedures. Check the **README** file provided with QuickBASIC 4.0 for further information.

3.4 C Calls to FORTRAN

This section applies the steps outlined in Section 3.1 to two examples of C-FORTRAN programs. A brief analysis follows each example.

3.4.1 Calling FORTRAN from C— Subroutine Call

The example below demonstrates a C main module calling a FORTRAN subroutine, `maxpar`. This subroutine adjusts the lower of two arguments to equal the higher argument.

■ Example

```
/* C source file - calls FORTRAN subroutine */
/* Compile in MEDIUM or LARGE model */

extern void fortran maxpar (int near *, int near *);

/* Declare as VOID, because there is no return value */
/* FORTRAN keyword causes C to use BASIC/FORTRAN/Pascal
   calling and naming conventions     */
/* Two integer params, passed by near reference */

main()
{
        int a = 5;
        int b = 7;

        printf("a = %d, b = %d", a, b);
        maxpar(&a, &b);
        printf("a = %d, b = %d", a, b);
}

C   FORTRAN source file, subroutine MAXPARAM
C
        SUBROUTINE MAXPARAM (I, J)
        INTEGER*2 I [NEAR]
        INTEGER*2 J [NEAR]
C
C   I and J received by near reference, because of NEAR attribute
C
        IF (I .GT. J) THEN
                J = I
        ELSE
                I = J
        ENDIF
        END
```

Naming conventions: The **fortran** keyword directs C to call `maxpar` with the BASIC/FORTRAN/Pascal naming convention (as MAXPAR). Note that `maxpar` is six letters long; it cannot be any longer because FORTRAN only recognizes the first six characters of any name.

Calling conventions: The **fortran** keyword directs C to call `maxpar` with the BASIC/FORTRAN/Pascal calling convention.

Parameter-passing methods: Since the FORTRAN subroutine `maxpar` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is specified in C by the use of near pointers, and in FORTRAN by applying the **NEAR** keyword to the parameter declarations.

Far reference could have been specified by using far pointers in C. In that case, the FORTRAN subroutine would not use the **NEAR** attribute (and would require the **FAR** attribute if compiled with medium memory model available with FORTRAN, Version 4.0).

3.4.2 Calling FORTRAN from C— Function Call

The example below demonstrates a C main module calling a FORTRAN function, `fact`. This function returns the factorial of an integer value.

■ Example

```
/* C source file - calls FORTRAN function */
/* Compile in MEDIUM or LARGE model */

extern int fortran fact (int);

/* FORTRAN keyword causes C to use BASIC/FORTRAN/Pascal
   calling and naming conventions     */
/* Integer parameter passed by value, the C default */

main()
{
        int x = 3;
        int y = 4;

        printf("The factorial of x   is %4d", fact(x));
        printf("The factorial of y   is %4d", fact(y));
        printf("The factorial of x+y is %4d", fact(x+y));
}
```

```
C   FORTRAN source file - factorial function
C
        INTEGER*2 FUNCTION FACT (N)
        INTEGER*2 N [VALUE]
C
C   N is received by value, because of VALUE attribute
C
        INTEGER*2 I
        FACT = 1
        DO 100 I = 1, N
            FACT = FACT * I
  100       CONTINUE
        RETURN
        END
```

Naming conventions: The **fortran** keyword directs C to call `fact` with the BASIC/FORTRAN/Pascal naming convention (as `FACT`). Note that word length is not an issue because `FACT` does not exceed six characters.

Calling conventions: The **fortran** keyword directs C to call `fact` with the BASIC/FORTRAN/Pascal calling convention.

Parameter-passing methods: When a parameter is passed that should not be changed, it is generally safest to pass the parameter by value. Passing by value is the default method in C, and is specified in FORTRAN by applying the **VALUE** attribute to the parameter declaration.

3.5 C Calls to Pascal

This section applies the steps outlined in Section 3.1 to two examples of C-Pascal programs. A brief analysis follows each example.

3.5.1 Calling Pascal from C—Procedure Call

The example below demonstrates a C main module calling a Pascal procedure, `maxpar`. This procedure adjusts the lower of two arguments to equal the higher argument.

- **Example**

```
/* C source file - calls Pascal procedure */
```

```
/* Compile in MEDIUM or LARGE model */

extern void pascal maxparam (int near *, int near *);

/* Declare as VOID, because there is no return value */
/* PASCAL keyword causes C to use BASIC/FORTRAN/Pascal
   calling and naming conventions     */
/* Two integer params, passed by near reference */

main()
{
        int a = 5;
        int b = 7;

        printf("a = %d, b = %d", a, b);
        maxparam(&a, &b);
        printf("a = %d, b = %d", a, b);
}

{ Pascal source code - Maxparam procedure. }

module Psub;
 procedure Maxparam(var a:integer; var b:integer);

{ Two integer parameters are received by near reference. }
{ Near reference is specified with the VAR keyword. }

 begin
    if a > b then
        b := a
    else
        a := b
 end;
end.
```

Naming conventions: The **pascal** keyword directs C to call `Maxparam` with the BASIC/FORTRAN/Pascal naming convention (as `MAXPARAM`).

Calling conventions: The **pascal** keyword directs C to call `Maxparam` with the BASIC/FORTRAN/Pascal naming convention.

Parameter-passing methods: Since the procedure `Maxparam` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference is used; this method is specified in C by the use of near pointers, and in Pascal with the **VAR** keyword.

Far reference could have been specified by using far pointers in C. In that case, the **VARS** keyword would be required instead of **VAR** in Pascal.

3.5.2 Calling Pascal from C—Function Call

The example below demonstrates a C main module calling a Pascal function, `fact`. This function returns the factorial of an integer value.

■ Example

```
/* C source file - calls Pascal function */
/* Compile in MEDIUM or LARGE model */

extern int pascal fact (int);

/* PASCAL keyword causes C to use BASIC/FORTRAN/Pascal
   calling and naming conventions     */
/* Integer parameter passed by value, the C default */

main()
{
        int x = 3;
        int y = 4;

        printf("The factorial of x   is %4d", fact(x));
        printf("The factorial of y   is %4d", fact(y));
        printf("The factorial of x+y is %4d", fact(x+y));
}
```

```
{ Pascal source code - factorial function. }

module Pfun;
 function Fact (n : integer) : integer;

{Integer parameters received by value, the Pascal default. }

 begin
    Fact := 1;
    while n > 0 do
        begin
            Fact := Fact * n;
            n := n - 1;           {Parameter n modified here.}
        end;
 end;
end.
```

Naming conventions: The **pascal** keyword directs C to call `fact` with the BASIC/FORTRAN/Pascal naming convention (as FACT).

Calling conventions: The **pascal** keyword directs C to call `fact` with the BASIC/FORTRAN/Pascal calling convention.

Parameter-passing methods: The Pascal function above should receive a parameter by value. Otherwise, the Pascal function will corrupt the parameter's value in the calling module. Passing by value is the default method for both C and Pascal.

CHAPTER 4

FORTRAN CALLS TO HIGH-LEVEL LANGUAGES

Microsoft FORTRAN supports calls to routines written in Microsoft C and Pascal. Also, if the main program is in BASIC, then a FORTRAN routine can call a BASIC routine. This chapter describes the necessary syntax for calling other languages from FORTRAN, and then gives examples for each language. Only simple parameter lists are used in this chapter.

For information on how to pass particular kinds of data, consult Part 2, "Data Handling Reference." Chapter 9 describes how to use the **VARYING** attribute with FORTRAN to pass a variable number of parameters.

4.1 The FORTRAN Interface to Other Languages

To call another language routine from within a FORTRAN function, first write an interface to the routine with the **INTERFACE** statement. This statement allows you to use special keywords (attributes) that affect how FORTRAN carries out calls. These keywords allow you to adjust naming conventions, calling conventions, and parameter-passing methods so that you can make routines from other languages compatible with FORTRAN.

4.1.1 The INTERFACE Statement

Here are the recommended rules for writing correct interfaces to routines from other languages:

1. Write an **INTERFACE** statement for each routine you call.

   Write the interface to a **FUNCTION** if the routine returns a value, or to a **SUBROUTINE** if the routine does not return a value. The **INTERFACE** statement should precede any calls to the routine.

2. Apply the **C** attribute to the routine if it is written in C (unless the C module is compiled with **/Gc** or is modified with the **fortran** or **pascal** keyword).

   The **C** attribute causes the routine to be called with the C naming and calling conventions. It also changes the default parameter-passing method for all parameters to pass by value. To apply the **C** attribute, type **[C]** immediately after the name of the routine.

3. If the routine is called from Pascal, you may want to apply the **PASCAL** attribute to the routine; this keyword does not change calling or naming conventions, but changes the default parameter-passing method for all parameters to pass by value.

To apply the **PASCAL** attribute, type **[PASCAL]** immediately after the name of the routine.

4. If the name of the routine is longer than six characters, use the **ALIAS** attribute. The use of **ALIAS** is explained in Section 4.1.2.

5. Adjust parameter-passing methods by applying the **VALUE**, **NEAR**, **FAR**, and **REFERENCE** attributes to parameter declarations.

   The **REFERENCE** attribute can be useful because the **C** and **PASCAL** keywords automatically change the default parameter-passing method to passing by value. For any given parameter, **REFERENCE** changes the method back to passing by reference. (By default, FORTRAN passes by far reference unless you are using medium memory model which is available with FORTRAN, Versions 4.0 and later.)

   To apply an attribute to a parameter declaration, put the attribute in brackets, along with any other attributes that modify the same object, and place the attribute(s) and brackets immediately after the parameter. (Refer to the examples below.)

6. Once the proper interface is set up, call the routine just as you would call any FORTRAN function or subroutine.

■ Examples

In the examples below, the variables `N`, `I`, and `J` are not significant, except for their types and attributes.

```
INTERFACE TO SUBROUTINE TEST [PASCAL] (N)
INTEGER*2 N [NEAR, REFERENCE]
END
```

The first example declares the subroutine `TEST` with the **PASCAL** attribute. This subroutine takes one argument, `N`, which has both the **NEAR** and **REFERENCE** attributes. `N` is passed by near reference.

```
INTERFACE TO FUNCTION REAL*8 CFUN [C] (I,J)
REAL*8 I [REFERENCE]
REAL*8 J
END
```

The second example declares a C function, `CFUN`, which returns a value of type **REAL*8**. The argument `I` is passed by far reference because of the **REFERENCE** attribute; the argument `J` is passed by value because the **C** attribute changes the default.

4.1.2 Using ALIAS

The **ALIAS** attribute is used with the following syntax

ALIAS:' *aliasname*'

where *aliasname* is the name that FORTRAN will actually place in the object code whenever the declared routine is called. When you use this feature, *aliasname* is precisely what FORTRAN will place in the object code; therefore, if you are linking to a C routine, be sure to type a leading underscore (_) before the name.

The **ALIAS** feature is necessary when the name of a routine is longer than 6 characters. Without **ALIAS**, FORTRAN would place only the first 6 characters into the object file, while the other language would place 7 or 8 characters into the other object file (or up to 40 in the case of BASIC). This difference would prevent the linker from finding a match.

- **Example**

```
INTERFACE TO PRINTNUM [C, ALIAS:'_printnum'] (N)
INTEGER*2 N
END
```

In the example above, **ALIAS** is used because `PRINTNUM` is longer than six characters. Note that the leading underscore should be used *only* when linking to a routine that uses the C naming convention.

4.2 Alternative FORTRAN Interface to C

Instead of modifying the behavior of FORTRAN with the **C** attribute, you can modify the behavior of C by applying the **pascal** or **fortran** keyword to the function definition heading. (These two keywords are functionally equivalent). Or you can compile the C module with the **/Gc** option, which specifies that all C functions, calls, and public symbols use the BASIC/FORTRAN/Pascal conventions.

For example, the following C function uses the BASIC/FORTRAN/Pascal conventions to receive an integer parameter:

```
int fortran fun1(n)
int n;
```

4.3 FORTRAN Calls to BASIC

Calls to BASIC from FORTRAN programs are not directly supported. No BASIC routine can be executed unless the main program is in BASIC, because a BASIC routine requires the environment to be initialized in a way that is unique to BASIC. No other language will perform this special initialization.

However, it is possible for a program to start up in BASIC, call a FORTRAN function that does most of the work of the program, and then call BASIC subprograms and function procedures as needed. Figure 4.1 illustrates how this can be done.

Here are the recommended rules for calling BASIC from FORTRAN:

1. Start up in a BASIC main module. You will need to use QuickBASIC, Version 4.0 or later, and may want to use the **DECLARE** statement to write an interface to the principal FORTRAN routine. (See Chapter 2, "BASIC Calls to High-Level Languages," for more information.)

2. Write an interface in FORTRAN for each BASIC routine you plan to call. Since BASIC and FORTRAN use the same basic calling convention, no special keyword is required to make FORTRAN compatible with BASIC.

3. Make sure that all data are passed as a near pointer. BASIC can pass data in a variety of ways, but is unable to receive data in any form other than near reference.

   With near pointers, the program assumes that the data are in the default data segment. If you want to pass data that are *not* in the default data segment (this is only a consideration with large model programs), then first copy the data to a variable that is in the default data segment.

■ Example

The example below demonstrates a BASIC program which calls a FORTRAN subroutine. The FORTRAN subroutine then calls a BASIC function that returns twice the number passed it, and a BASIC subprogram that prints two numbers.

```
' BASIC source
'
DEFINT A-Z
DECLARE SUB Fprog ()
CALL Fprog
END
'
FUNCTION Dbl(N) STATIC
    Dbl = N*2
END FUNCTION
'
SUB Printnum(A,B) STATIC
    PRINT "The first number is ";A
    PRINT "The second number is ";B
END SUB
```

```
C       FORTRAN subroutine
C        Calls a BASIC function that receives one integer,
C          and a BASIC subprogram that takes two integers.
C
        INTERFACE TO INTEGER*2 FUNCTION DBL (N)
        INTEGER*2 N [NEAR]
        END
C
C       ALIAS attribute necessary because BASIC recognizes more
C         than six characters of the name "Printnum"
C
        INTERFACE TO SUBROUTINE PRINTN [ALIAS:'Printnum'] (N1, N2)
        INTEGER*2 N1 [NEAR]
        INTEGER*2 N2 [NEAR]
        END
C
C       Parameters must be declared NEAR in the parameter
C         declarations; BASIC receives ONLY 2-byte pointers
C
        SUBROUTINE FPROG
        INTEGER*2 DBL
        INTEGER*2 A,B
        A = 5
        B = 6
        WRITE (*,*) 'Twice of 5 is ', DBL(A)
        CALL PRINTN(A,B)
        END
```

In the example above, note that the **NEAR** attribute is used in the FORTRAN routines, so that near addresses will be passed to BASIC instead of far addresses.

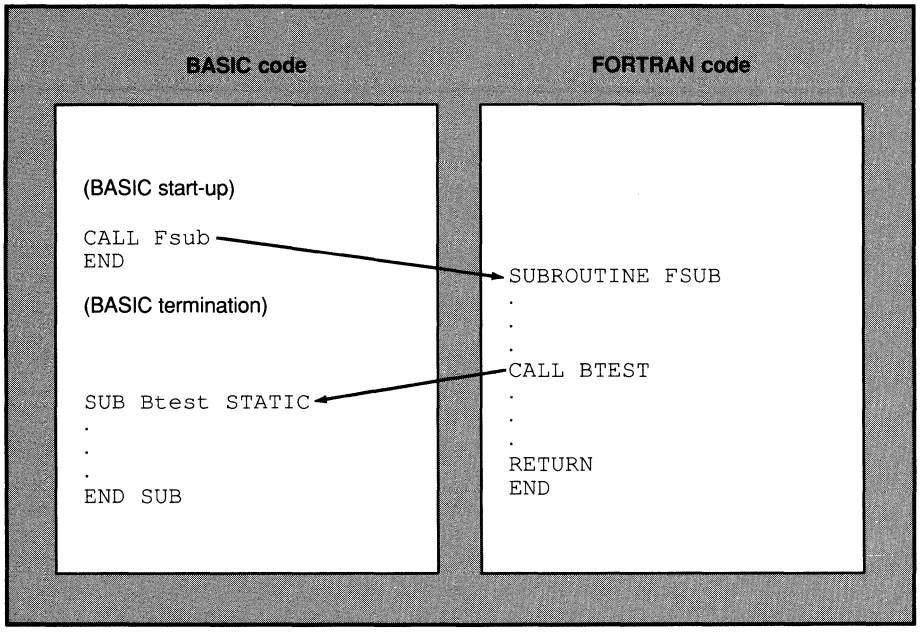


Figure 4.1 FORTRAN Call to BASIC

4.4 FORTRAN Calls to C

Writing FORTRAN interfaces to C is fairly straightforward; however, if you are using Version 4.0 of FORTRAN, then linking requires some additional steps, which are described in Chapter 1.

This section applies the steps outlined in Section 4.1 to two examples of FORTRAN-C programs. A brief analysis follows each example.

4.4.1 Calling C from FORTRAN— No Return Value

The example below demonstrates a FORTRAN main module calling a C function, `maxparam`. This function returns no value but adjusts the lower of two parameters to equal the higher argument.

■ Example

```
C   FORTRAN SOURCE FILE - CALLS C FUNCTION, NO RETURN VALUE
C
        INTERFACE TO SUBROUTINE MAXPARAM[C,ALIAS:'_maxparam'](I,J)
        INTEGER*2 I [NEAR, REFERENCE]
        INTEGER*2 J [NEAR, REFERENCE]
        END
C
C   C ATTRIBUTE DIRECTS FORTRAN TO USE C CONVENTIONS
C   ALIAS NECESSARY BECAUSE 'MAXPARAM' LONGER THAN 6 CHARS.
C   EACH PARAMETER PASSED BY NEAR REFERENCE
C
        INTEGER*2 I,J
        I = 5
        J = 7
        WRITE (*,*) 'I = ',I,' J = ', J
        CALL MAXPARAM(I,J)
        WRITE (*,*) 'I = ',I,' J = ', J
        END
```

```
/* C source file */
/* Compile in MEDIUM or LARGE memory model */
/* Maxparam declared VOID because no return value */

void maxparam(p1, p2)
int near *p1;       /* Integer params received by near ref. */
int near *p2;       /* NEAR keyword not needed in MEDIUM model */
{
        if (*p1 > *p2)
                *p2 = *p1;
        else
                *p1 = *p2;
}
```

Naming conventions: By default, FORTRAN only places the first six characters of MAXPARAM into the object file, whereas C places all eight. This conflict is resolved with the **ALIAS** attribute: both modules place _maxparam (consistent with the C naming convention) into an object file.

Calling conventions: The **C** attribute (in the **INTERFACE** statement) causes MAXPARAM to be called with the C calling convention, which pushes parameters in reverse order and specifies other lower-level differences.

Parameter-passing methods: Since the function maxparam may alter the value of either parameter, both must be passed by reference. Near reference is implemented in FORTRAN with the **NEAR** and **REFERENCE** attributes, and in C by using near pointers. The **REFERENCE**

attribute is necessary in FORTRAN because the C keyword changes the default passing method to pass by value.

Far reference could have been specified by leaving off the **NEAR** keyword from the FORTRAN parameter declarations. In that case, the C module would need to use far pointers.

4.4.2 Calling C from FORTRAN— Function Call

The example below demonstrates a FORTRAN main module calling a C function, `fact`. This function returns the factorial of an integer value.

■ Example

```
C   FORTRAN SOURCE FILE - CALLS C FUNCTION
C
        INTERFACE TO INTEGER*2 FUNCTION FACT [C] (N)
        INTEGER*2 N
        END
C
C   C ATTRIBUTE DIRECTS FORTRAN TO USE C CONVENTIONS
C   PARAMETER PASSED BY VALUE, WHICH IS DEFAULT WHEN
C      C ATTRIBUTE IS IN USE
C
        INTEGER*2 FACT
        INTEGER*2 I,J
        I = 3
        J = 4
        WRITE (*,*) 'The factorial of I   is ',FACT(I)
        WRITE (*,*) 'The factorial of J   is ',FACT(J)
        WRITE (*,*) 'The factorial of I+J is ',FACT(I+J)
        END
```

```
/* C source file */
/* Compile in MEDIUM or LARGE model */
/* Factorial function, returning integer */

int fact(n)
int n;          /* Integer received by value, the C default */
{
        int     result = 1;

        while (n)
                result *= n--;  /* Parameter n modified here */
        return (result);
}
```

Naming conventions: The **C** attribute (in the **INTERFACE** statement) causes FACT to be called with the C naming convention (as `_fact`). Word length is not a concern; `fact` does not exceed six characters.

Calling conventions: The **C** attribute (in the **INTERFACE** statement) causes FACT to be called with the C calling convention, which pushes parameters in reverse order and specifies other lower-level differences.

Parameter-passing methods: The C function above should receive the parameter by value. Otherwise, the function will corrupt the parameter's value in the calling module. Passing by value is the default method for C; it is also the default method for FORTRAN whenever the **C** attribute is in use.

4.5 FORTRAN Calls to Pascal

Calling Pascal from FORTRAN is usually fairly simple, because the **PASCAL** attribute causes FORTRAN to use the Pascal default of passing data by value.

This section applies the steps outlined in Section 4.1 to two examples of FORTRAN-Pascal programs. A brief analysis follows each example.

4.5.1 Calling Pascal from FORTRAN— Procedure Call

The example below demonstrates a FORTRAN main module calling a Pascal procedure, Maxparam. This procedure adjusts the lower of two parameters to equal the higher argument.

■ Example

```
C   FORTRAN SOURCE FILE - CALLS PASCAL PROCEDURE
C
        INTERFACE TO SUBROUTINE MAXPARAM [ALIAS:'MAXPARAM'] (I,J)
        INTEGER*2 I [NEAR]
        INTEGER*2 J [NEAR]
        END
C
C   ALIAS NECESSARY BECAUSE 'MAXPARAM' LONGER THAN 6 CHARS.
C   EACH PARAMETER PASSED BY NEAR REFERENCE
C
        INTEGER*2 I,J
        I = 5
        J = 7
        WRITE (*,*) 'I = ',I,' J = ', J
        CALL MAXPARAM(I,J)
        WRITE (*,*) 'I = ',I,' J = ', J
        END
```

```
{ Pascal source code - Maxparam procedure. }

module Psub;
 procedure Maxparam(var a:integer; var b:integer);

{ Two integer parameters are received by near reference. }
{ Near reference is specified with the VAR keyword. }

 begin
    if a > b then
        b := a
    else
        a := b
 end;
end.
```

Naming conventions: By default, FORTRAN only places the first six characters of MAXPARAM into the object file, whereas Pascal places all eight. The **ALIAS** attribute resolves this conflict: both modules place MAXPARAM into an object file.

Calling conventions: FORTRAN and Pascal use the same convention for calling.

Parameter-passing methods: Since the procedure `Maxparam` may alter the value of either parameter, both must be passed by reference. Near reference was implemented in FORTRAN with the **NEAR** attributes, and in Pascal with the **VAR** keyword. The **PASCAL** attribute is not used here because no parameter is being passed by value.

Far reference could have been specified by leaving off the **NEAR** keyword from the FORTRAN parameter declarations. In that case, the Pascal module would use **VARS** instead of **VAR**.

4.5.2 Calling Pascal from FORTRAN— Function Call

The example below demonstrates a FORTRAN main module calling a Pascal function, `Fact`. This function returns the factorial of an integer value.

■ Example

```
C   FORTRAN SOURCE FILE - CALLS PASCAL FUNCTION
C
        INTERFACE TO INTEGER*2 FUNCTION FACT [PASCAL] (N)
        INTEGER*2 N
        END
```

```
C
C   PARAMETER PASSED BY VALUE, WHICH IS DEFAULT WHEN
C     PASCAL ATTRIBUTE IS IN USE
C
        INTEGER*2 FACT
        INTEGER*2 I,J
        I = 3
        J = 4
        WRITE (*,*) 'The factorial of I   is ',FACT(I)
        WRITE (*,*) 'The factorial of J   is ',FACT(J)
        WRITE (*,*) 'The factorial of I+J is ',FACT(I+J)
        END
```

```
{ Pascal source code - factorial function. }

module Pfun;
 function Fact (n : integer) : integer;

{Integer parameters received by value, the Pascal default. }

 begin
    Fact := 1;
    while n > 0 do
        begin
            Fact := Fact * n;
            n := n - 1;           {Parameter n modified here.}
        end;
 end;
end.
```

Naming conventions: FORTRAN and Pascal use a similar naming convention. The **ALIAS** attribute is not necessary because the function name does not exceed six characters.

Calling conventions: FORTRAN and Pascal use the same calling convention.

Parameter-passing methods: The Pascal function above should receive the parameter by value. Otherwise, the function will corrupt the parameter's value in the calling module. Passing by value is the default method for Pascal; it is also the default method for FORTRAN whenever the **PASCAL** attribute is in use.

CHAPTER 5

PASCAL CALLS TO HIGH-LEVEL LANGUAGES

Microsoft Pascal supports calls to routines written in Microsoft FORTRAN and C. Also, if the main program is in BASIC, then a Pascal routine can call a BASIC routine. This chapter describes the necessary syntax for calling these other languages, and then gives examples for each language. Only simple parameter lists are used in this chapter.

For information on how to pass particular kinds of data, consult Part 2, "Data Handling Reference." Chapter 9 describes how to use the **VARYING** attribute with Pascal to pass a varying number of parameters.

5.1 The Pascal Interface to Other Languages

You can provide an interface from Pascal to a routine in a different programming language. This interface is created by writing an **extern** function or procedure declaration. This declaration informs Pascal that the routine is to be found in another module; furthermore, you can use special keywords with the declaration to affect how Pascal makes calls to the routine. These keywords allow you to adjust naming conventions, calling conventions, and parameter-passing methods, so that the other language routines will be compatible with Pascal.

Here are the recommended steps for writing an **extern** declaration:

1. Declare a function (for routines that return values) or a procedure (for routines that do not); all normal rules of Pascal syntax apply to the heading. Instead of writing a procedure body, however, simply type the word **extern**, followed by a semicolon (;).

   The **extern** can be placed in the procedure declaration section of any Pascal function or procedure that needs to call the different language routine.

2. If you are calling a C function, attach the **C** attribute to the declaration. To use this attribute, type **C** in brackets, at the very end of the function or procedure heading (immediately before the semicolon).

   This attribute directs Pascal to use the C naming and calling conventions. There is no similar keyword for FORTRAN or BASIC; they use the same naming and calling conventions used by Pascal.

3. Decide how you want to pass each parameter. By default Pascal passes parameters by value. The **VAR** keyword, applied to individual parameters, specifies passing by near reference, and the **VARS** keyword specifies passing by far reference.

4. Once the routine is properly declared, call it just as though it were a Pascal function or procedure.

■ **Examples**

```
procedure Calc(var i:integer; x:real) [C]; extern;
```

In the example above, the **C** attribute directs Pascal to use the C calling and naming conventions.

```
function Quadratic(a,b,c : integer) : real [C]; extern;
```

In this example also, the **C** attribute directs Pascal to use the C calling and naming conventions.

```
procedure Total (a,b,c : integer, var sum : integer); extern;
```

The third example, by default, uses the BASIC/FORTRAN/Pascal standard naming and calling conventions.

5.2 Alternative Pascal Interface to C

Instead of modifying the behavior of Pascal with the **C** attribute, you can modify the behavior of C by applying the **pascal** or **fortran** keyword to the function definition heading. (These two keywords are functionally equivalent.) You can also compile the C module with the **/Gc** option, which specifies that all C functions, calls, and public symbols use the BASIC/FORTRAN/Pascal conventions.

■ **Example**

```
int pascal fun1(n)
int n;
```

In the example above, the C function uses the BASIC/FORTRAN/Pascal conventions to receive an integer parameter.

5.3 Pascal Calls to BASIC

Calls to BASIC from Pascal programs are not directly supported. No BASIC routine can be executed unless the main program is in BASIC because a BASIC routine requires the environment to be initialized in a unique way. No other language will perform this special initialization.

However, it is possible for a program to start up in BASIC, call a Pascal routine that does most of the work of the program, and then call BASIC subprograms and function procedures as needed. The following diagram illustrates how this can be done:

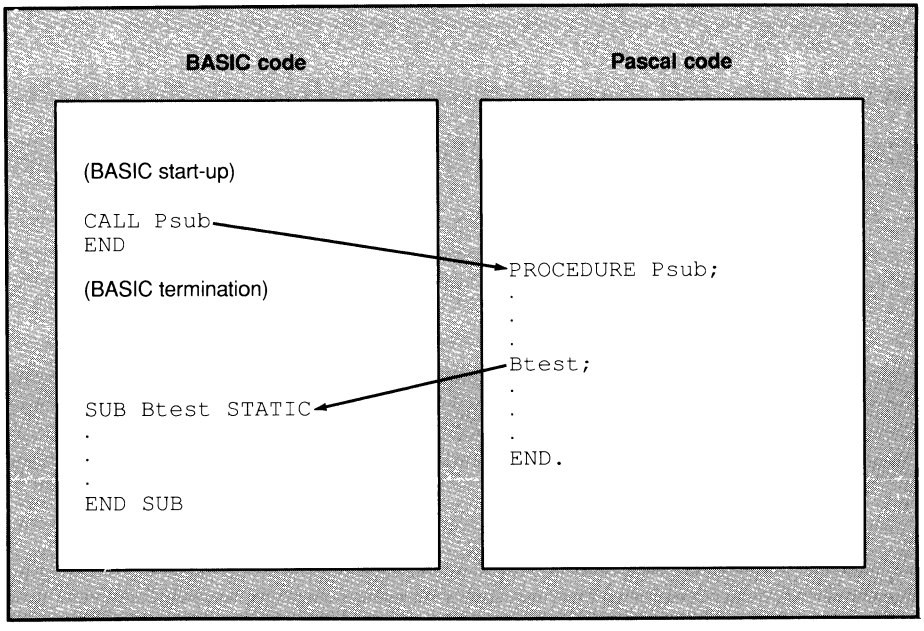


Figure 5.1 Pascal Call to BASIC

Observe the following steps when calling BASIC from Pascal:

1. Start up in a BASIC main module. You will need to use QuickBASIC, Version 4.0 or later, and may want to use the **DECLARE** statement to write an interface to the principal Pascal routine. (See Chapter 2, "BASIC Calls to High-Level Languages," for additional information.)

2. Once in Pascal code, declare each BASIC routine you plan to call, in an **extern** procedure or function declaration.

3. Make sure that all data are passed as a near pointer, by declaring each argument with **VAR**. BASIC can pass data in a variety of ways, but is unable to receive data in any form other than as a near pointer.

With near pointers, the program assumes that the data are in the default data segment. If you want to pass data that are *not* in the default data segment (this is only a consideration with far-heap allocation), then first copy the data to a variable that is in the default data segment.

■ Example

The following example demonstrates a BASIC program which calls a Pascal procedure. The Pascal procedure then calls a BASIC function that returns twice the number passed it, and a BASIC subprogram that prints two numbers.

```
' BASIC source
'
DEFINT A-Z
DECLARE SUB Pprog ()
CALL Pprog
END
'
FUNCTION Dbl(N) STATIC
    Dbl = N*2
END FUNCTION
'
SUB Printnum(A,B) STATIC
    PRINT "The first number is ";A
    PRINT "The second number is ";B
END SUB


{* Pascal procedure *}
{* Calls a BASIC function and a BASIC subprogram *}

module pproc;
procedure Pprog();

function Dbl (var n:integer):integer; extern;
procedure Printnum (var n1,n2:integer); extern;
var a,b:integer;

    begin
        a := 5;
        b := 6;
        writeln ('Twice of 5 is ', Dbl(a));
        Printnum(a,b);
    end;
end.
```

In the example above, note that every argument in the external declarations must be declared **VAR**, since BASIC can only receive near pointers as parameters.

5.4 Pascal Calls to C

This section applies the steps outlined in Section 5.1 to two examples of Pascal-C programs. A brief analysis follows each example.

5.4.1 Calling C from Pascal—No Return Value

The example below demonstrates a Pascal main module calling a C function, maxparam. This function returns no value, but adjusts the lower of two arguments to equal the higher.

■ Example

```
{ Pascal source file - calls C function, no return value }

program Pcsub (input, output);
procedure Maxparam (var i,j : integer) [C]; extern;

{ C attribute directs Pascal to use C conventions. }
{ VAR indicates each parameter passed by near reference. }

var
    a, b : integer;
begin
    a := 5;
    b := 7;
    writeln('a = ',a,'b = ',b);
    Maxparam(a,b);
    writeln('a = ',a,'b = ',b);
end.
```

```
/* C source file */
/* Compile in MEDIUM or LARGE memory model */
/* Maxparam declared VOID because no return value */

void maxparam(p1, p2)
int near *p1;       /* Integer params received by near ref. */
int near *p2;       /* NEAR keyword not needed in MEDIUM model */
{
        if (*p1 > *p2)
                *p2 = *p1;
        else
                *p1 = *p2;
}
```

Naming conventions: The C attribute causes Maxparam to be called with the C naming convention (as _maxparam).

Calling conventions: The **C** attribute causes `Maxparam` to be called with the C calling convention, which pushes parameters in reverse order.

Parameter-passing methods: Since the subprogram `Maxparam` may alter the value of either parameter, both arguments must be passed by reference. In this case, near reference was chosen; this method is specified in Pascal with the **VAR** keyword, and in C by using near pointers.

Far reference could have been specified by using **VARS** instead of **VAR**; in that case, the C parameter declarations would use far pointers.

5.4.2 Calling C from Pascal—Function Call

The example below demonstrates a Pascal main module calling a C function, `fact`. This function returns the factorial of an integer value.

■ Example

```
{ Pascal source file - calls C function }

program Pcfun (input, output);
function Fact (n : integer) [C]; extern;

{ C attribute directs Pascal to use C conventions. }
{ Parameter passed by value, the default method. }

var
    a, b : integer;
begin
    a := 3;
    b := 4;
    writeln('The factorial of a   is ', Fact(a));
    writeln('The factorial of b   is ', Fact(b));
    writeln('The factorial of a+b is ', Fact(a+b));
end.
```

```
/* C source file */
/* Compile in MEDIUM or LARGE model */
/* Factorial function, returning integer */

int fact(n)
int n;          /* Integer received by value, the C default */
{
        int     result = 1;

        while (n)
                result *= n--;  /* Parameter n modified here */
        return (result);
}
```

Naming conventions: The **C** attribute causes `Fact` to be called with the C naming convention (as `_fact`).

Calling conventions: The **C** attribute causes `Fact` to be called with the C calling convention, which pushes parameters in reverse order, and specifies other low-level differences.

Parameter-passing methods: The C function should receive the integer parameter by value. Otherwise, the function will corrupt the value of the parameter in the calling routine. Passing by value is the default method for both Pascal and C.

5.5 Pascal Calls to FORTRAN

This section applies the steps outlined in Section 5.1 to two examples of Pascal-FORTRAN programs. A brief analysis follows each example.

5.5.1 Calling FORTRAN from Pascal— Subroutine Call

The example below demonstrates a Pascal main module calling a FORTRAN subroutine, MAXPARAM. This subroutine adjusts the lower of two arguments to equal the higher.

■ **Example**

```
{ Pascal source file - calls FORTRAN subroutine }

program Pfsub (output);
procedure Maxpar (var i,j : integer) ; extern;

{ Name must not exceed six characters. }
{ VAR indicates each parameter passed by near reference. }

var
    a, b : integer;
begin
    a := 5;
    b := 7;
    writeln('a = ',a,'b = ',b);
    Maxpar(a,b);
    writeln('a = ',a,'b = ',b);
end.
```

```
C   FORTRAN source file, subroutine MAXPARAM
C
        SUBROUTINE MAXPARAM (I, J)
        INTEGER*2 I [NEAR]
        INTEGER*2 J [NEAR]
C
C   I and J received by near reference, because of NEAR attribute
C
        IF (I .GT. J) THEN
                J = I
        ELSE
                I = J
        ENDIF
        END
```

Naming conventions: By default, Pascal places all eight characters of `Maxparam` into the object file, whereas FORTRAN places only the first six. This conflict is resolved by shortening the name of the Pascal routine to six characters.

Calling conventions: Pascal and FORTRAN use the same calling convention.

Parameter-passing methods: Since the subprogram `Maxparam` may alter the value of either parameter, both arguments must be passed by reference. In this case, near reference was chosen; this method is specified in Pascal with the **VAR** keyword, and in FORTRAN by applying the **NEAR** attribute to each parameter declaration.

Far reference could have been chosen by using **VARS** instead of **VAR**. In that case, the **NEAR** attribute would not be used in the FORTRAN parameter declarations.

5.5.2 Calling FORTRAN from Pascal—Function Call

The example below demonstrates a Pascal main module calling a FORTRAN function, `FACT`. This function returns the factorial of an integer value.

■ Example

```
{ Pascal source file - calls FORTRAN function }

program Pffun (output);
function Fact (n : integer); extern;

{ Parameter passed by value, the default method. }

var
    a, b : integer;
begin
    a := 3;
    b := 4;
    writeln('The factorial of a   is ', Fact(a));
    writeln('The factorial of b   is ', Fact(b));
    writeln('The factorial of a+b is ', Fact(a+b));
end.
```

```
C   FORTRAN source file - factorial function
C
        INTEGER*2 FUNCTION FACT (N)
        INTEGER*2 N [VALUE]
C
C   N is received by value, because of VALUE attribute
C
        INTEGER*2 I
        FACT = 1
        DO 100 I = 1, N
            FACT = FACT * I
  100       CONTINUE
        RETURN
        END
```

Naming conventions: There are no conflicts with naming conventions, because the function name (FACT) does not exceed six characters.

Calling conventions: Pascal and FORTRAN use the same convention for calling.

Parameter-passing methods: When passing a parameter that should not be changed, it is generally safest to pass the parameter by value. Passing by value is the default method in Pascal, and is specified in FORTRAN by applying the **VALUE** attribute to a parameter declaration.

CHAPTER 6

ASSEMBLY-TO-HIGH-LEVEL INTERFACE

With the Microsoft Macro Assembler you can write assembly modules that can be linked to modules developed with Microsoft BASIC, C, Pascal, or FORTRAN. This chapter first outlines the recommended programming guidelines for writing assembly routines compatible with Microsoft high-level languages; it then gives examples specific to each language.

Writing assembly routines for Microsoft high-level languages is easiest when you use the simplified segment directives provided with the Macro Assembler, Version 5.0. In general, this manual assumes that you have Version 5.0. For information on writing assembly-language interfaces without the simplified segment directives, turn to Section 6.7 in order to look up **SEGMENT**, **GROUP**, and **ASSUME** statements.

6.1 Writing the Assembly Procedure

The Microsoft BASIC, C, FORTRAN, and Pascal compilers use roughly the same interface for procedure calls. This section describes the interface, so that you can call assembly procedures using essentially the same methods as Microsoft compiler-generated code. Procedures written with these methods can be called recursively and can be effectively used with the Stack Trace feature of the Microsoft CodeView® debugger.

The standard assembly-interface method consists of these steps:

- Setting up the procedure
- Entering the procedure
- Allocating local data (optional)
- Preserving register values
- Accessing parameters
- Returning a value (optional)
- Exiting the procedure

Sections 6.1.1–6.1.7 describe each of these steps.

6.1.1 Setting Up the Procedure

The linker cannot combine the assembly procedure with the calling program unless compatible segments are used and unless the procedure itself is declared properly. The following points may be helpful:

1. Use the **.MODEL** directive at the beginning of the source file, if you have Version 5.0 of the Macro Assembler; this directive automatically causes the appropriate kind of returns to be

generated (**NEAR** for small or compact model, **FAR** otherwise). Modules called from Pascal should be declared as **.MODEL LARGE**; modules called from BASIC should be **.MODEL MEDIUM**. If you have a version of the assembler previous to 5.0, declare the procedure **FAR** (or **NEAR** if the calling program is small- or compact-model C).

2. If you have Version 5.0 or later of the Microsoft Macro Assembler, use the simplified segment directives **.CODE** to declare the code segment and **.DATA** to declare the data segment. (Having a code segment is sufficient if you do not have data declarations.) If you are using an earlier version of the assembler, look up **SEGMENT**, **GROUP**, and **ASSUME** directives in Section 6.7, "The Microsoft Segment Model."
3. The procedure label must be declared public with the **PUBLIC** directive. This declaration makes the procedure available to be called by other modules. Also, any data you want to make public to other modules must be declared as **PUBLIC**.
4. Global data or procedures accessed by the routine must be declared **EXTRN**. The safest way to use **EXTRN** is to place the directive outside of any segment definition (however, near data should generally go inside the data segment).

6.1.2 Entering the Procedure

Two instructions begin the procedure:

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

This sequence establishes **BP** as the "framepointer." The framepointer is used to access parameters and local data, which are located on the stack. **SP** cannot be used for this purpose because it is not an index or base register. Also, the value of **SP** may change as more data are pushed onto the stack. However, the value of the base register **BP** will remain constant throughout the procedure, so that each parameter can be addressed as a fixed displacement off of **BP**.

The instruction sequence above first saves the value of **BP**, since it will be needed by the calling procedure as soon as the current procedure terminates. Then **BP** is loaded with the value of **SP** in order to capture the value of the stack pointer at the time of entry to the procedure.

6.1.3 Allocating Local Data (Optional)

An assembly procedure can use the same technique for implementing local data that is used by high-level languages. To set up local data space, simply decrease the contents of **SP** in the third instruction of the procedure. (To ensure correct execution, you should always increase or decrease **SP** by an even amount.) Decreasing **SP** reserves space on the stack for the local data. The space must be restored at the end of the procedure.

```
push    bp
mov     bp,sp
sub     sp,space
```

In the text above, *space* is the total size in bytes of the local data. Local variables are then accessed as fixed, negative displacements off of **BP**.

- **Example**

```
push    bp
mov     bp,sp
sub     sp,4
.
.
.
mov     WORD PTR [bp-2],0
mov     WORD PTR [bp-4],0
```

The example above uses two local variables, each of which is two bytes in size. **SP** is decreased by 4, since there are four bytes total of local data. Later, each of the variables is initialized to 0. These variables are never formally declared with any assembler directive; the programmer must keep track of them manually.

Local variables are also called dynamic, stack, or automatic variables.

6.1.4 Preserving Register Values

A procedure called from any of the Microsoft high-level languages should preserve the values of **SI**, **DI**, **SS**, and **DS** (in addition to **BP**, which is already saved). Therefore, push any of these register values that the procedure alters. If the procedure does not change the value of any of these registers, then the registers do not need to be pushed.

The recommended method (used by the high-level languages) is to save registers after the framepointer is set and local data (if any) are allocated.

```
push    bp              ; Save old framepointer
mov     bp,sp           ; Establish current framepointer
sub     sp,4            ; Allocate local data space
push    si              ; Save SI and DI
push    di
.
.
.
```

In the example above, **DI** and **SI** (in that order) must be popped before the end of the procedure.

6.1.5 Accessing Parameters

Once you have established the procedure's framepointer, allocated local data space (if desired), and pushed any registers that need to be preserved, you can write the main body of the procedure. In order to write instructions that can access parameters, consider the general picture of the stack frame after a procedure call as illustrated in Figure 6.1.

The stack frame for the procedure is established by the following sequence of events:

1. The calling program pushes each of the parameters on the stack, after which **SP** points to the last parameter pushed.
2. The calling program issues a **CALL** instruction, which causes the return address (the place in the calling program to which control will ultimately return) to be placed on the stack. This address may be either two bytes long (for near calls) or four bytes long (for far calls). **SP** now points to this address.
3. The first instruction of the called procedure saves the old value of **BP**, with the instruction `push bp`. **SP** now points to the saved copy of **BP**.
4. **BP** is used to capture the current value of **SP**, with the instruction `mov bp,sp`. **BP** therefore now points to the old value of **BP**.
5. Whereas **BP** remains constant throughout the procedure, **SP** may be decreased to provide room on the stack, for local data or saved registers.

In general, the displacement (off of **BP**) for a parameter X is equal to:

```
2 + size of return address
  + total size of parameters between X and BP
```

For example, consider a **FAR** procedure that has received one parameter, a two-byte address. The displacement of the parameter would be:

```
Argument's displacement =  2 + size of return address
                        =  2 + 4
                        =  6
```

The argument can thus be loaded into **BX** with the following instruction:

```
        mov     bx, [bp+6]
```

Once you determine the displacement of each parameter, you may want to use string equates or structures so that the parameter can be referenced with a single identifier name in your assembly source code. For example, the parameter above at **BP+6** can be conveniently accessed if you put the following statement at the beginning of the assembly source file:

```
Arg1    EQU     [bp+6]
```

You could then refer to this parameter as `Arg1` in any instruction. Use of this feature is optional.

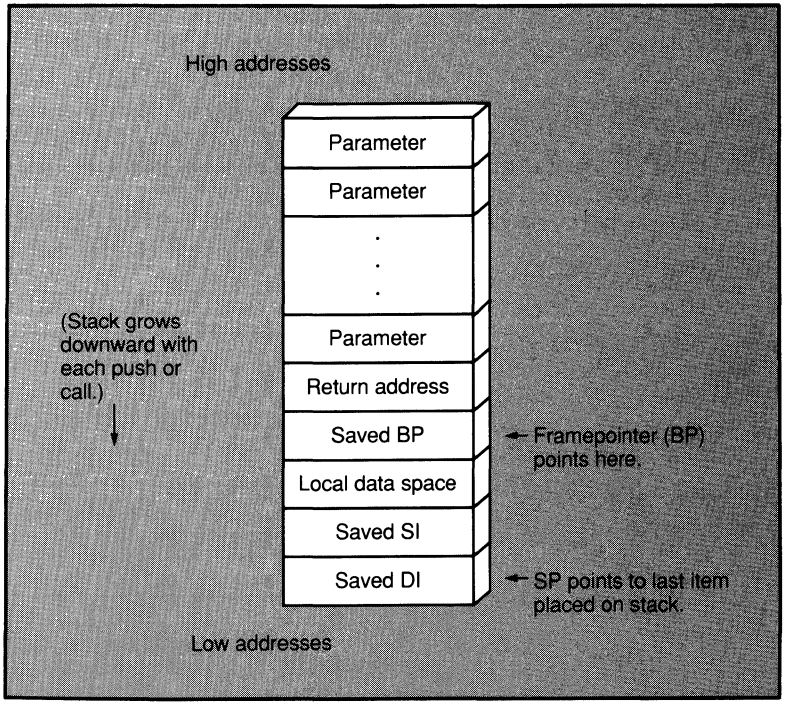


Figure 6.1 The Stack Frame

Note

Microsoft high-level languages always push segment addresses before pushing offset address. Furthermore, when pushing arguments larger than two bytes, high-order words are always pushed before low-order words.

This standard for pushing segment addresses before pushing offset addresses facilitates the use of the **LES** instruction, as demonstrated in Section 6.4, "Calls from FORTRAN."

6.1.6 Returning a Value (Optional)

Microsoft BASIC, C, FORTRAN, and Pascal share similar conventions for receiving return values. The conventions are the same when the data type to be returned is simple (that is, not an array or structured type) and is no more than four bytes long. This includes all **NEAR** and **FAR** address types (in other words, all pointers and all parameters passed by reference).

Data size	Returned in register
1 byte	**AL**
2 bytes	**AX**
4 bytes	High-order portion (or segment address) in **DX**; low-order portion (or offset address) in **AX**

When the return value is larger than four bytes, a procedure called by C must allocate space for the return value and then place its address in **DX**:**AX**. A convenient way to create space for the return value is to simply declare it in a data segment.

If your assembly procedure is called by BASIC, FORTRAN or Pascal, then it must use a special convention in order to return floating-point values, records, user-defined types and arrays, and values larger than four bytes. This convention is presented below.

■ BASIC/FORTRAN/Pascal Long Return Values

In order to create an interface for long return values, BASIC, FORTRAN and Pascal modules take the following actions before they call your procedure:

1. First they create space, somewhere in the stack segment, to hold the actual return value.
2. When the call to your procedure is made, an extra parameter is passed; this parameter contains the offset address of the actual return value. This parameter is placed immediately above the return address. (In other words, this parameter is the last one pushed.)
3. The segment address of the return value is contained in both **SS** and **DS**.

The extra parameter (which contains the offset address of the return value) is always located at **BP+6**. Furthermore, its presence automatically increases the displacement of all other parameters by two, as shown in the following comparison:

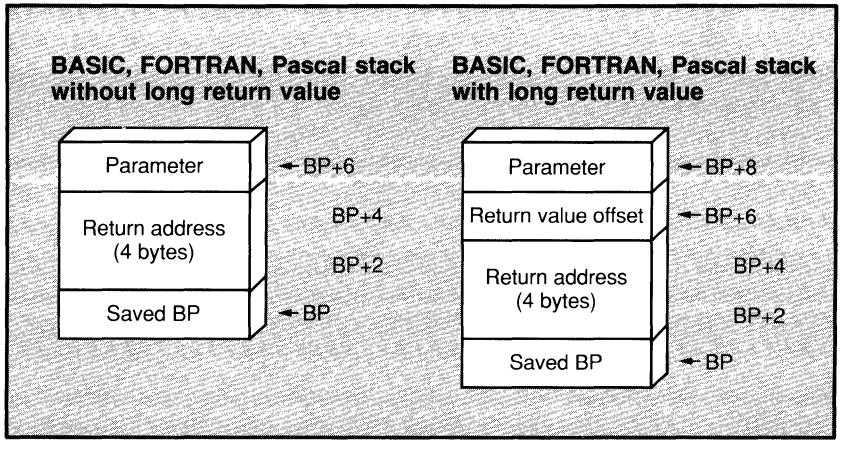


Figure 6.2 BASIC/FORTRAN/Pascal Long Return Values

Your assembly procedure will successfully return a long value if you follow these steps:

1. Put the data for the return value at the location pointed to by the return value offset.
2. Copy the return-value offset (located at **BP+6**) to **AX**, and copy **SS** to **DX**. This is necessary because the calling module expects **DX:AX** to point to the return value.
3. Exit the procedure as described in the next section.

6.1.7 Exiting the Procedure

Several steps may be involved in terminating the procedure:

1. If any of the registers **SS**, **DS**, **SI**, or **DI** have been saved, these must be popped off the stack in the reverse order that they were saved.
2. If local data space was allocated at the beginning of the procedure, **SP** must be restored with the instruction `mov sp,bp`.
3. Restore **BP** with `pop bp`. This step is always necessary.
4. Finally, return to the calling program with `ret`. If the BASIC, FORTRAN, or Pascal calling convention is in use, then use the `ret` *n* form of the instruction to adjust the stack with respect to the parameters that were pushed by the caller. (If the procedure is called by a C module, then the calling module will perform this adjustment.)

■ **Examples**

```
pop     bp
ret
```

The example above shows the simplest possible exit sequence. No registers were saved, no local data space was allocated, and the C calling convention is in use.

```
pop     di              ; Pop saved regs
pop     si
mov     sp,bp           ; Remove local data space
pop     bp              ; Restore old framepointer
ret     6               ; Exit, and restore 6 byte of args
```

The example above shows an exit sequence for a procedure that has previously saved **SI** and **DI**, allocated local data space, and uses a non-C calling convention. The procedure must therefore use `ret 6` to restore the six bytes of parameters on the stack.

6.2 Calls from BASIC

A BASIC program can call an assembly procedure in another source file with the use of the **CALL**, **CALLS**, or **DECLARE** statement. In addition to the steps outlined in Section 6.1, "Writing the Assembly Procedure," the following guidelines may be helpful:

1. Declare procedures called from BASIC as **FAR**.

2. Observe the BASIC calling convention.

   a. Upon exit, the procedure must reset **SP** to the value it had before the parameters were placed on the stack. This is accomplished with the instruction `ret` *size*, where *size* is the total size in bytes of all the parameters.

   b. Parameters are placed on the stack in the same order in which they appear in the BASIC source code. The first parameter will be highest in memory (because it is also the first parameter to be placed on the stack, and the stack grows downward).

   c. By default, BASIC parameters are passed by reference as two-byte addresses.

3. Observe the BASIC naming convention.

   BASIC outputs symbolic names in uppercase characters, which is also the default behavior of the assembler. BASIC recognizes up to 40 characters of a name, whereas the assembler recognizes only the first 31, but this should rarely create a problem.

In the following example program, QuickBASIC 4.0 calls an assembly procedure that calculates "A x 2^B," where A and B are the first and second parameters, respectively. The calculation is performed by shifting the bits in A to the left, B times. (Note: with earlier versions of BASIC, you need to rewrite the example so that it calls a subprogram, not a function.)

```
DEFINT A-Z
PRINT "3 times 2 to the power of 5 is ";
PRINT Power2(3,5)
END
```

To understand how to write the assembly procedure, consider how the parameters are placed on the stack:

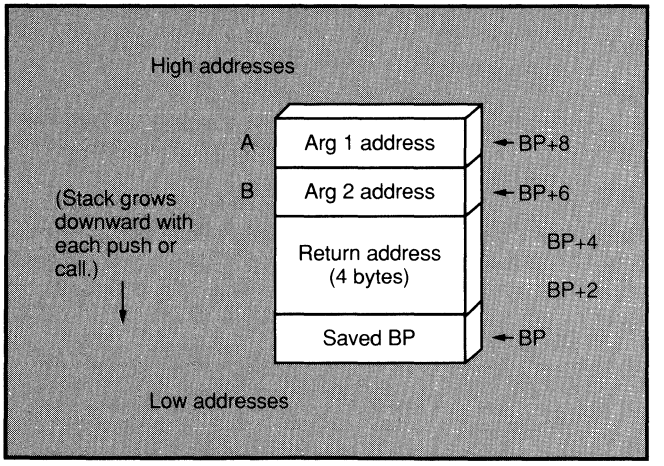


Figure 6.3 BASIC Stack Frame

The return address is four bytes long because procedures called from BASIC must be **FAR**. Arg 1 (parameter 1) is higher in memory than Arg 2 because BASIC pushes arguments (parameters) in the same order in which they appear. Also, each argument is passed as a two-byte offset address, the BASIC default.

The assembly procedure can be written as follows:

```
.MODEL MEDIUM
.CODE
        PUBLIC  Power2
Power2  PROC
        push    bp              ; Entry sequence - saved old BP
        mov     bp,sp           ; Set stack framepointer

        mov     bx,[bp+8]       ; Load Arg1 into
        mov     ax,[bx]         ;   AX
        mov     bx,[bp+6]       ; Load Arg2 into
        mov     cx,[bx]         ;   CX
        shl     ax,cl           ; AX = AX * (2 to power of CX)
                                ; Leave return value in AX

        pop     bp              ; Exit sequence - restore old BP
        ret     4               ; Return, and restore 4 bytes
Power2  ENDP
        END
```

Note that each parameter must be loaded in a two-step process because the *address* of each is passed rather than the value. Also, note that the stack is restored with the instruction `ret 4` since the total size of the parameters is four bytes.

6.3 Calls from C

A C program can call an assembly procedure in another module, just as it would call a C function. In addition to the steps outlined in Section 6.1, "Writing the Assembly Procedure," the following guidelines may prove helpful:

1. Declare procedures called from C as **FAR** if the C module is compiled in large, huge, or medium model, and **NEAR** if the C module is compiled in small or compact model (although the **near** and **far** keywords can override these defaults). The correct declaration for the procedure is made implicitly when you use the **.MODEL** directive available in the Microsoft Macro Assembler, Version 5.0.

2. Observe the C calling convention.

   a. Return with a simple `ret` instruction. Do *not* restore the stack with `ret` *size*, since the calling C routine will restore the stack itself, as soon as it resumes control.

   b. Parameters are placed on the stack in the reverse order that they appear in the C source code. The first parameter will be lowest in memory (because it is the last parameter to placed on the stack, and the stack grows downward).

   c. By default, C parameters are passed by value, except for arrays, which are passed by reference.

3. Observe the C naming convention.

   Include an underscore in front of any name which will be shared publicly with C. C recognizes only the first eight characters of any name, so do not make names shared with C longer than eight characters. Also, if you plan to link with the **/NOIGNORECASE** option, remember that C is case sensitive and does not convert names to uppercase. Assemble with the **/MX** option to prevent **MASM** from converting names to uppercase.

In the following example program, C calls an assembly procedure that calculates "A x 2^B," where A and B are the first and second parameters, respectively. The calculation is performed by shifting the bits in A to the left, B times.

The C program uses an **extern** declaration to create an interface with the assembly procedure. No special keywords are required because the assembly procedure will use the C calling convention.

```
extern int power2(int, int);

main()
{
    printf("3 times 2 to the power of 5 is %d\n", power2(3,5));
}
```

To understand how to write the assembly procedure, consider how the parameters are placed on the stack, as illustrated in Figure 6.4.

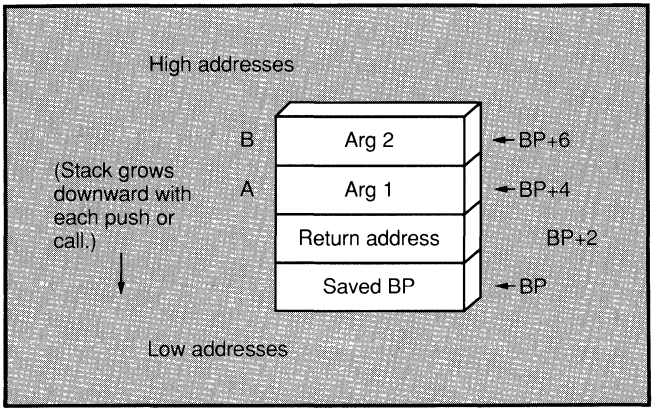


Figure 6.4 C Stack Frame

The return address is two bytes long, assuming that the C module is compiled in small or compact model. If the C module is compiled in large, huge, or medium model, then the addresses of Arg 1 and Arg 2 are each increased by two, to BP+6 and BP+8, respectively, because the return address will be four bytes long.

Arg 1 (parameter 1) is lower in memory than Arg 2, because C pushes arguments in the reverse order that they appear. Each argument is passed by value.

The assembly procedure can be written as follows:

```
.MODEL SMALL
.CODE
        PUBLIC  _power2
_power2 PROC
        push    bp              ; Entry sequence - save old BP
        mov     bp,sp           ; Set stack framepointer

        mov     ax,[bp+4]       ; Load Arg1 into AX
        mov     cx,[bp+6]       ; Load Arg2 into CX
        shl     ax,cl           ; AX = AX * (2 to power of CX)
                                ; Leave return value in AX

        pop     bp              ; Exit sequence - restore old BP
        ret                     ; Return
_power2 ENDP
        END
```

The example above assumes that the C module is compiled in small model. The parameter offsets and the **.MODEL** directive will change for different models.

Note that `ret` without a size variable is used, since the caller will adjust the stack upon return from the call.

6.4 Calls from FORTRAN

A FORTRAN program can call an external assembly procedure with the use of the **INTERFACE** statement. However, the **INTERFACE** statement is not strictly necessary unless you intend to change one of the FORTRAN defaults. In addition to the steps outlined in Section 6.1, "Writing the Assembly Procedure," the following guidelines may be helpful:

1. Declare procedures called from FORTRAN as **FAR**.

2. Observe the FORTRAN calling convention.

   a. Upon exit, the procedure must reset **SP** to the value it had before the arguments were placed on the stack. This is accomplished with the instruction `ret` *size*, where *size* is the total size of all the parameters.

b. Arguments are placed on the stack in the same order in which they appear in the FORTRAN source code. The first parameter will be highest in memory (because it is also the first parameter to be placed on the stack, and the stack grows downward).

c. By default, FORTRAN parameters are passed by reference as far addresses if the FORTRAN module is compiled in large or huge memory model, and as near addresses if the FORTRAN module is compiled in medium model. Versions of FORTRAN prior to Version 4.0 are always large model.

3. Observe the FORTRAN naming convention.

   FORTRAN only recognizes the first 6 characters of any name, while the assembler recognizes the first 31. Names shared publicly with FORTRAN should not be longer than 6 characters, unless the FORTRAN module is using the **ALIAS** feature.

In the following example, FORTRAN calls an assembly procedure that calculates "A x 2^B," where A and B are the first and second parameters, respectively. This is done by shifting the bits in A to the left, B times.

The FORTRAN module uses the **INTERFACE** statement, which is described in Section 4.1, "The FORTRAN Interface to Other Languages."

```
      INTERFACE TO INTEGER*2 POWER2(A,B)
      INTEGER*2 A,B
      END
C
      INTEGER*2 A,B
      A = 3
      B = 5
      WRITE (*,*) '3 times 2 to the power of 5 is ',POWER2(A,B)
      END
```

To understand how to write the assembly procedure, consider how the parameters are placed on the stack, as illustrated in Figure 6.5.

Figure 6.5 assumes large-model FORTRAN. If you compile the FORTRAN module in medium model, then each argument will be passed as a two-byte, not four-byte address. The return address is four bytes long because procedures called from FORTRAN must always be **FAR**.

Arg 1 (parameter 1) is higher in memory than Arg 2 because FORTRAN pushes arguments (parameters) in the same order that they appear.

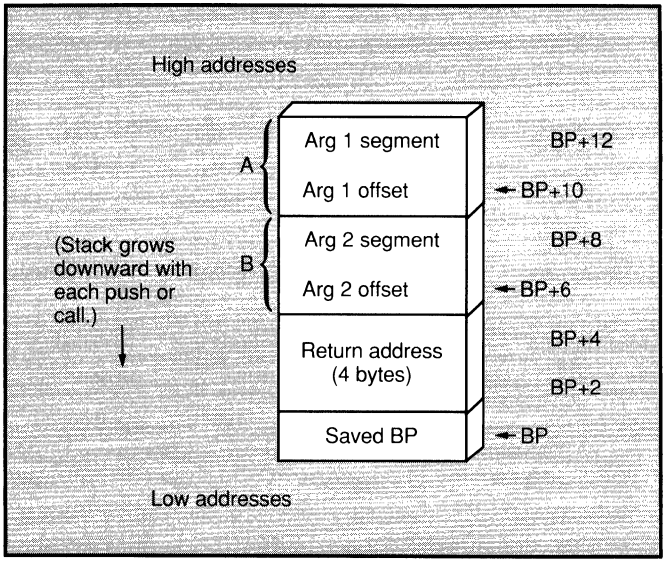


Figure 6.5 FORTRAN Stack Frame

The assembly procedure can be written as follows:

```
.MODEL LARGE
.CODE
        PUBLIC  Power2
Power2  PROC
        push    bp              ; Entry sequence - save old BP
        mov     bp,sp           ; Set stack framepointer

        les     bx,[bp+10]      ; Load Arg1 into
        mov     ax,es:[bx]      ;   AX
        les     bx,[bp+6]       ; Load Arg2 into
        mov     cx,es:[bx]      ;   CX
        shl     ax,cl           ; AX = AX * (2 to power of CX)
                                ; Leave return value in AX

        pop     bp              ; Exit sequence - restore old BP
        ret     8               ; Return and restore 8 bytes
Power2  ENDP
        END
```

In the example above, each argument must be loaded using the four-byte address that was pushed onto the stack. The procedure loads four-byte addresses with the **LES** instruction, which loads the destination operand (in this case, **BX**) with the source operand, and also loads **ES** with the object two bytes higher in memory. Thus, the instruction

```
les     bx,[bp+10]
```

loads **BX** with the value at `BP+10` (an offset address), and **ES** with the value at `BP+12` (a segment address), which is necessary to set up the next instruction.

Upon exit, the stack is restored with the instruction `ret 8`, since the total size of parameters pushed onto the stack is eight.

6.5 Calls from Pascal

A Pascal program can call an assembly procedure in another module just as it would call a Pascal routine. In addition to the steps outlined in Section 6.1, "Writing the Assembly Procedure," the following guidelines may be helpful:

1. Declare procedures called from Pascal as **FAR**. This is taken care of for you automatically if you use the **MODEL** directive available with the Microsoft Macro Assembler, Version 5.0 or later; specify **LARGE**.

2. Observe the Pascal calling convention.

   a. Upon exit, the procedure must reset **SP** to the value it had before the parameters were placed on the stack. The procedure resets **SP** with the instruction `ret` *size*, where *size* is the total size of all the parameters pushed on the stack.

   b. Parameters are placed on the stack in the same order in which they appear in the Pascal source code. The first parameter will be highest in memory (because it is also the first parameter to be placed on the stack, and the stack grows downward.)

   c. By default, Pascal parameters are passed by value.

3. Observe the Pascal naming convention.

Pascal only recognizes the first 8 characters of any name, while the assembler recognizes the first 31. Names shared publicly with Pascal should not be longer than 8 characters.

In the following example program, Pascal calls an assembly procedure that calculates "A x 2^B," where A and B are the first and second parameters, respectively. The calculation is performed by shifting the bits in A to the left, B times.

The Pascal module uses an **extern** declaration in its interface with the assembly procedure. No special keywords are required, because the assembly procedure will use the Pascal calling convention.

```
program Asmtest(input, output);
function Power2(a,b:integer):integer; extern;
begin
    writeln('3 times 2 to the power of 5 is ',Power2(3,5));
end.
```

To understand how to write the assembly procedure, consider how the parameters are placed on the stack, as illustrated in Figure 6.6.

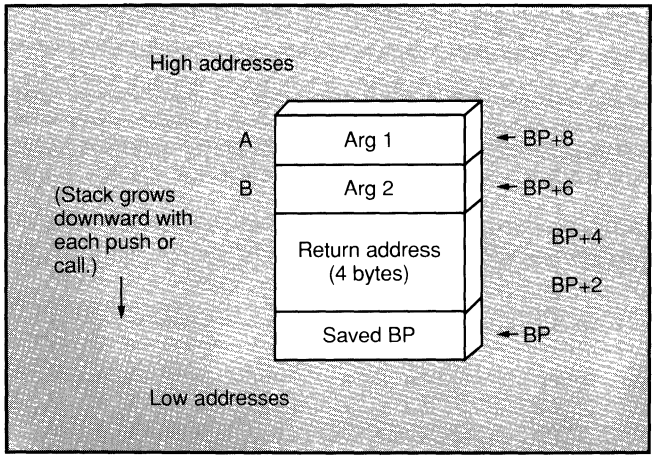


Figure 6.6 Pascal Stack Frame

Arg 1 (parameter 1) is higher in memory than Arg 2 because Pascal pushes arguments in the same order that they appear. Each argument is passed by value.

The assembly procedure can be written as follows:

```
.MODEL LARGE
.CODE
        PUBLIC  Power2
Power2  PROC
        push    bp              ; Entry sequence - save old BP
        mov     bp,sp           ; Set stack framepointer

        mov     ax,[bp+8]       ; Load Arg1 into AX
        mov     cx,[bp+6]       ; Load Arg2 into CX
        shl     ax,cl           ; AX = AX * (2 to power of CX)
                                ; Leave return value in AX

        pop     bp              ; Exit sequence - restore old BP
        ret     4               ; Return and restore 4 bytes
Power2  ENDP
        END
```

The **AX** and **CX** registers can be loaded directly because the parameters were passed by value. Note that the `ret 4` instruction is necessary to clear the stack of the four bytes of parameters.

6.6 Calling High-Level Languages from Assembly

High-level language routines assume that certain initialization code has previously been executed; you can ensure that the proper initialization is performed by starting in a high-level language module, and then calling an assembly procedure. The assembly procedure can then call high-level language routines as needed, as shown in Figure 6.7.

To execute an assembly call to a high-level language, you need to observe the following guidelines:

1. Push each parameter onto the stack, observing the calling convention of the high-level language. Constants such as offset addresses must first be loaded into a register before being pushed.
2. With long parameters, always push the segment or high-order portion of the parameter first, regardless of the calling convention.
3. If you are using the BASIC/FORTRAN/Pascal calling convention with a function that returns a non integer value, then allocate an additional two-byte parameter. This additional parameter should contain the offset of the location where you want the value returned, and must be pushed onto the stack last.
4. Execute a call. The call must be far unless the high-level-language routine is small model.

5. If the routine used the C calling convention, then immediately after the call you must clear the stack of parameters with the instruction

   ```
   add sp,size
   ```

   where *size* is the total size in bytes of all parameters that were pushed.

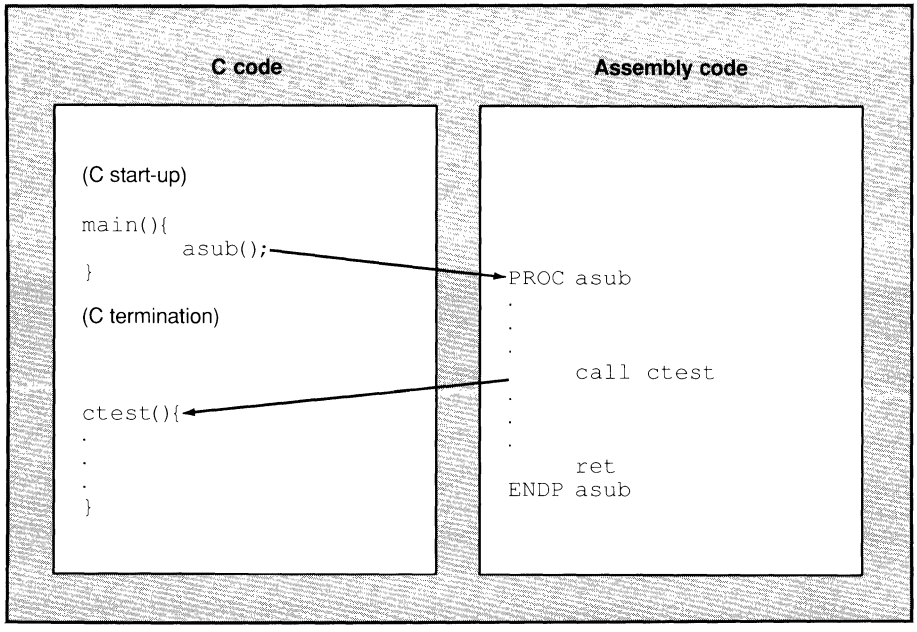


Figure 6.7 Assembly Call to C

6.7 The Microsoft Segment Model

If you use the simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, versions of the Macro Assembler prior to 5.0 do not support these directives. With older versions of the assembler, you should use the **SEGMENT**, **GROUP**, **ASSUME**, and **ENDS** directives equivalent to the simplified segment directives.

Table 6.1 shows the default segment names created by each directive. Use of these segments ensures compatibility with Microsoft languages and will help you to access public symbols. This table is followed by a list of three steps, illustrating how to make the actual declarations, and an example program.

Table 6.1

Default Segments and Types for Standard Memory Models

Model	Directive	Name	Align	Combine	Class	Group
Small	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Medium	.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Compact	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Large	.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP

The directives in Table 6.1 refer to the following kinds of segments:

Directive	Description of Segment
.CODE	The segment containing all the code for the module.
.DATA	Initialized data.
.DATA?	Uninitialized data. Microsoft compilers store uninitialized data separately because it can be more efficiently stored than initialized data.
.FARDATA and **.FARDATA?**	Data placed here will not be combined with the corresponding segments in other modules. The segment of data placed here can always be determined, however, with the assembler **SEG** operator.
.CONST	Constant data. Microsoft compilers use this segment for such items as string and floating-point constants.
.STACK	Stack. Normally, this segment is declared in the main module for you and should not be redeclared.

The following steps describe how to use Table 6.1 to create directives:

1. Determine what memory model you are using. Then refer to Table 6.1 to look up the segment name, align type, combine type, and class for your code and data segments. Use all of these attributes when you define a segment. For example, the code segment for small model is declared as follows:

   ```
   _TEXT          SEGMENT   WORD PUBLIC 'CODE'
   ```

   The name _TEXT and all the attributes are taken from Table 6.1. If the combine type is private, simply do not use any combine type.

2. If you have segments in **DGROUP**, put them into **DGROUP** with the **GROUP** directive, as in:

   ```
   GROUP          DGROUP    _DATA _BSS
   ```

3. Use **ASSUME** and **ENDS** as you would normally. Upon entry, **DS** and **SS** will both point to **DGROUP**; therefore, a small-model procedure that makes use of **DGROUP** should include the following **ASSUME** directive:

   ```
   ASSUME         CS:TEXT, DS:DGROUP, SS:DGROUP
   ```

   A large-model procedure will assume a different code segment, and may assume a far data segment for **ES**.

The following example shows the C-assembly program from Section 6.3, without the simplified segment directives from Version 5.0 of the Microsoft Macro Assembler:

```
_TEXT   SEGMENT WORD PUBLIC 'CODE'
        ASSUME  cs:_TEXT
        PUBLIC  _Power2
_Power2 PROC
        push    bp          ; Entry sequence - save BP
        mov     bp,sp       ; Set stack frame

        mov     ax,[bp+4]   ; Load Arg1 into AX
        mov     cx,[bp+6]   ; Load Arg2 into CX
        shl     ax,cl       ; AX = AX * (2 to power of CX)
                            ; Leave return value in AX

        pop     bp          ; Exit sequence - restore BP
        ret                 ; Return
_Power2 ENDP
_TEXT   ENDS
        END
```

DATA HANDLING REFERENCE

CHAPTERS

PART 2 ◇ DATA HANDLING REFERENCE

Part 2 explains how to pass types of data, with focus on those types of data (such as strings of text) that are stored in a different format by each language. This part also summarizes parameter-passing methods and describes alternative methods for sharing data between modules.

CHAPTER 7

PASSING BY REFERENCE OR VALUE

Chapter 2 introduced the general concepts of passing by reference and passing by value. Chapter 2 also listed the default method used by each language. For example, BASIC passes by reference, and Pascal passes by value.

This chapter describes features in each language that override the default. For example, using the **BYVAL** keyword in a **DECLARE** statement will cause BASIC to pass a given parameter by value rather than by reference.

This chapter is divided into four sections, each of which summarizes parameter-passing methods in a particular language, discussing how to pass arguments by value, by near reference, and by far reference. To write a successful mixed-language interface, you must consider how each parameter is passed by the calling routine and how each is received by the called routine.

7.1 BASIC Arguments

The default for BASIC is to pass all arguments by near reference.

Note

Every BASIC subprogram or function always *receives* parameters by near reference. The rest of this section describes how BASIC *passes* parameters only.

■ Passing BASIC Arguments by Value

An argument is passed by value when the called routine is first declared with a **DECLARE** statement, and the **BYVAL** keyword is applied to the argument. The use of **CALLS** overrides this default and passes by far reference instead, as mentioned below.

■ Passing BASIC Arguments by Near Reference

The BASIC default is to pass by near reference. Use of **SEG**, **BYVAL**, or **CALLS** changes this default.

■ Passing BASIC Arguments by Far Reference

BASIC will pass each argument in a call by far reference when **CALLS** is used to invoke a routine. Using **SEG** to modify a parameter in a preceding **DECLARE** statement will also cause BASIC to pass that parameter by far reference.

■ Examples

```
DECLARE SUB Test(BYVAL a%, b%, SEG c%)
.
.
.
CALL Test(x%, y%, z%)
```

The example above passes the first argument (a%) by value, the second argument (b%) by near reference, and the third argument (c%) by far reference.

```
CALLS Test2(x%, y%, z%)
```

The example above passes each argument by far reference.

7.2 C Arguments

The default for C is to pass all arrays by reference (near or far, depending on the memory model) and all other data types by value. C uses far data pointers for compact, large, and huge model, and near data pointers for small and medium model.

■ Passing C Arguments by Value

The C default is to pass all nonarrays (which includes all data types other than those explicitly declared as arrays) by value.

Arrays can be passed by value by being declared as the only member of a structure. The following example passes all 100 bytes of `x` directly to the function `test()`.

```
struct x_struct {int x[100]} xs;
.
.
.
test(xs);
```

The function `test`, in turn, receives the array by declaring a parameter of type `x_struct`. A C routine would interpret this data object as a structure, so that structure syntax would be used to manipulate an array element, as in the following example:

```
test(x_arrs)
struct  x_struct        x_arrs;
{
        x_arrs.x[0] = 1;  /* set first element to 1 */
```

Routines written in other languages, however, would not require structure or record syntax. FORTRAN, for example, would access the first element simply as `X(1)`.

■ Passing C Arguments by Reference (Near or Far)

In C, passing a pointer to an object is equivalent to passing the object itself by reference. After control is passed to the called function, each reference to the parameter itself is prefixed by *.

Note

To pass a pointer to a object, prefix the parameter in the call statement with &. To receive a pointer to an object, prefix the parameter's declaration with *. In the latter case, this may mean adding a second * to a parameter which already has a *. For example, to receive a pointer by value, declare it as

```
int         *ptr;
```

but to receive the same pointer by reference, declare it as

```
int         **ptr;
```

The default for arrays is to pass by reference.

■ Effect of Memory Models on Size of Reference

Near reference is the default for passing pointers in small and medium model C. Far reference is the default in the compact, large, and huge models.

Near pointers can be specified with the **near** keyword, which overrides the default pointer size. However, if you are going to override the default pointer size of a parameter, then you must explicitly declare the parameter type in function declarations as well as function definitions.

Far pointers can be specified with the **far** keyword, which overrides the default pointer size.

7.3 FORTRAN Arguments

The FORTRAN default is to pass and receive all arguments by reference. The size of the address passed depends on the memory model.

■ Passing FORTRAN Arguments by Value

A parameter is passed by value when declared with the **VALUE** attribute. This declaration can occur either in a FORTRAN **INTERFACE** statement (which determines how to pass a parameter) or in a function or subroutine declaration (which determines how to receive a parameter).

A function or subroutine declared with the **PASCAL** or **C** attribute will pass by value all parameters declared in its parameter list (except for parameters declared with the **REFERENCE** attribute). This change in default passing method applies to function and subroutine definitions, as well as to an **INTERFACE** statement.

■ Passing FORTRAN Arguments by Reference (Near or Far)

Passing by reference is the default for FORTRAN. However, if either the **C** or **PASCAL** attribute is applied to a function or subroutine declaration, then you need to apply the **REFERENCE** attribute to any parameter of the routine that you want passed by reference.

■ Use of Memory Models and FORTRAN Reference Parameters

Near reference is the default for medium-model FORTRAN programs; far reference is the default for large-model and huge-model programs.

Note

Versions of FORTRAN prior to 4.0 always compile in large memory model.

You can apply the **NEAR** attribute to reference parameters in order to specify near reference. You can apply the **FAR** attribute to reference parameters in order to specify far reference. These keywords enable you to override the default. They have no effect when they specify the same method as the default.

You may need to apply more than one attribute to a given parameter. In that case, enclose both attributes in brackets, separated by a comma:

```
REAL*4 X [NEAR, REFERENCE]
```

7.4 Pascal Arguments

The Pascal default is to pass all arguments by value.

■ Passing Pascal Arguments by Near Reference

Parameters are passed by near reference when declared as **VAR** or **CONST**.

Parameters are also passed by near reference when the **ADR** of a variable, or a pointer to a variable, is passed by value. In other words, the address of the variable is first determined. Then, this address is passed by value. (This is essentially the same method employed in C.)

■ Passing Pascal Arguments by Far Reference

Parameters are passed by far reference when declared as **VARS** or **CONSTS**.

Parameters are also passed by far reference when the **ADRS** of a variable is passed by value.

CHAPTER 8

NUMERICAL, LOGICAL, AND STRING DATA

This chapter considers the details of passing and receiving kinds of data. Discussion focuses on the differences in string format and on the methods of passing strings between each combination of languages.

8.1 Integer and Real Numbers

Integers and reals are usually the simplest kinds of data to pass between languages. However, the type of numerical data is named differently in each language; furthermore, not all data types are available in every language, and another type may have to be substituted in some cases.

Table 8.1 shows equivalent data types in each language.

Warning

As noted in Table 8.1, C sometimes performs automatic data conversions which the other languages do not perform. You can prevent C from performing such conversions by declaring a variable as the only member of a structure and then passing this structure. For example, you can pass a variable *x* of type **float**, by first declaring the structure:

```
struct {
          float   x;
} x_struct;
```

If you pass a variable of type **char** or **float** by value and do not take this precaution, then the C conversion may cause the program to fail.

8.2 FORTRAN COMPLEX Types

The FORTRAN types **COMPLEX*8** and **COMPLEX*16** are not directly implemented in any other language. However, you can write structures in C, records in Pascal, and user-defined types in BASIC that are precisely equivalent.

The type **COMPLEX*8** has two fields: the first is a four-byte floating-point number that contains the real component, and the second is a four-byte floating point number that contains the imaginary component.

Table 8.1
Equivalent Numeric Data Types

BASIC	C	FORTRAN	Pascal
x%	**short**	INTEGER*2	INTEGER2
INTEGER	**int**		INTEGER (default)
...	**unsigned short***	...	WORD
	unsigned		
x&	**long**	INTEGER*4	INTEGER4
LONG		INTEGER (default)	
...	**unsigned long***	...	...
x!	**float**†	REAL*4	REAL4
x (default)		REAL	REAL (default)
SINGLE			
x#	**double**	REAL*8	REAL8
DOUBLE		DOUBLE PRECISION	
...	**unsigned char**‡	CHARACTER*1§	
	BOOLEAN		

* Not available in BASIC, FORTRAN, or Pascal. A signed integer may be substituted, but take care not to exceed range.

† C automatically converts **float** to **double** in assignment or when passed by value.

‡ C automatically converts **char** and **unsigned char** to **int** in assignment or when passed by value.

§ The FORTRAN type **CHARACTER*1** is *not* the same as **LOGICAL**. The data type **LOGICAL** is covered in Section 8.3.

The type **COMPLEX*16** is similar to **COMPLEX*8**, with the only difference being that each field contains an eight-byte floating-point number.

The type **COMPLEX** is equivalent to the type **COMPLEX*8.**

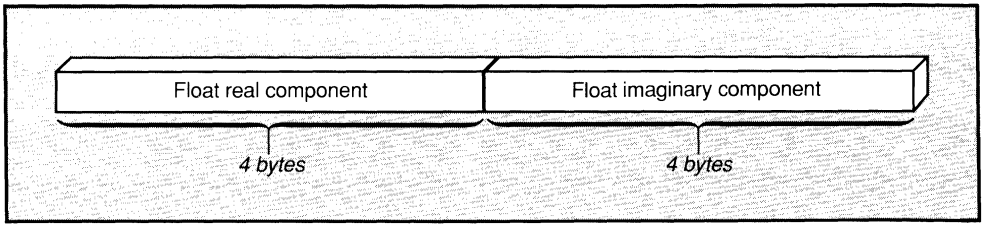


Figure 8.1 FORTRAN COMPLEX Data Format

8.3 FORTRAN LOGICAL Type

The FORTRAN **LOGICAL** type is not equivalent to either the Pascal **BOOLEAN** or C **char** type. Instead, a FORTRAN **LOGICAL*2** is stored as a one-byte indicator value (1=true, 0=false) followed by an unused byte. A FORTRAN **LOGICAL*4** is stored as a one-byte indicator value followed by three unused bytes. The type **LOGICAL** is equivalent to **LOGICAL*4**, unless **$STORAGE:2** is in effect.

To pass or receive a FORTRAN **LOGICAL** type, declare a C structure, Pascal record, or BASIC user-defined type, with the appropriate fields.

8.4 Strings

Strings are stored in a variety of formats. Therefore, some transformation is frequently required to pass strings between languages.

This section presents the string format(s) used in each language, and then describes methods for passing strings within specific combinations of languages.

8.4.1 String Formats

The following section describes how a string is stored by each language, as well as how a string is passed as an argument.

■ BASIC String Format

Strings are stored in BASIC as four-byte string descriptors:

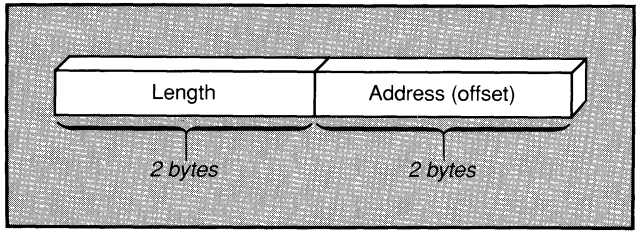


Figure 8.2 BASIC String Descriptor Format

The first field of the string descriptor contains a two-byte integer indicating the length of the actual string text. The second field contains the address of this text. This address is an offset into the default data area and is assigned by BASIC's string-space management routines. These management routines need to be available to reassign this address whenever the length of the string changes, yet these management routines are only available to BASIC. Therefore, other languages should not alter the length of a BASIC string.

■ C String Format

C stores strings as simple arrays of bytes and uses a null character (numerical 0, ASCII NUL) as delimiter. For example, consider the string declared as follows:

```
char str[] = "String of text"
```

The string is stored in 15 bytes of memory as:

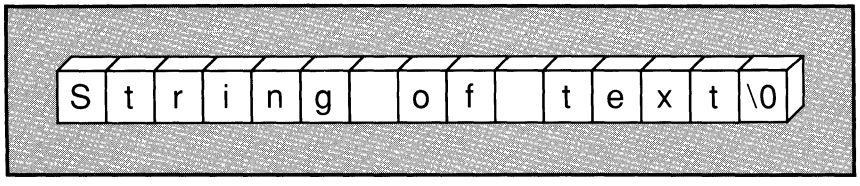


Figure 8.3 C String Format

Since `str` is an array like any other, it is passed by reference, just as other C arrays are. To pass by value, declare the array as a member of a structure. (See Section 7.2, "C Arguments," for more information.)

■ FORTRAN String Format

FORTRAN stores strings as a series of bytes at a fixed location in memory. There is no delimiter at the end of the string as in C. Consider the string declared as follows:

```
STR = 'String of text'
```

The string is stored in 14 bytes of memory as:

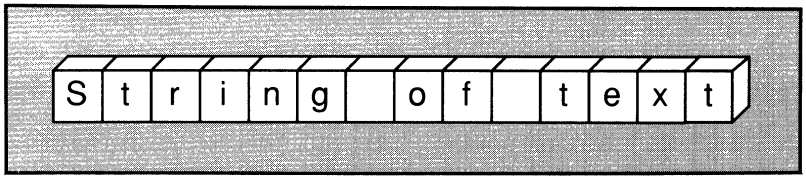


Figure 8.4 FORTRAN String Format

Strings are passed by reference, just as other FORTRAN data are. Although Version 4.0 of the FORTRAN Optimizing Compiler has a method for passing length, the variable length FORTRAN strings *cannot* be used in a mixed-language interface because other languages cannot access the temporary variable that FORTRAN uses to communicate string length.

■ Pascal String Format

Pascal has two types of strings, each of which uses a different format: a fixed-length type **STRING** and the variable-length type **LSTRING**.

The format used for **STRING** is identical to the FORTRAN string format, described above.

The format of an **LSTRING** stores the length in the first byte. For example, consider an **LSTRING** declared as:

```
VAR STR:LSTRING(14);
STR := 'String of text'
```

The string is stored in 15 bytes of memory. The first byte indicates the length of the string text. The remaining bytes contain the string text itself:

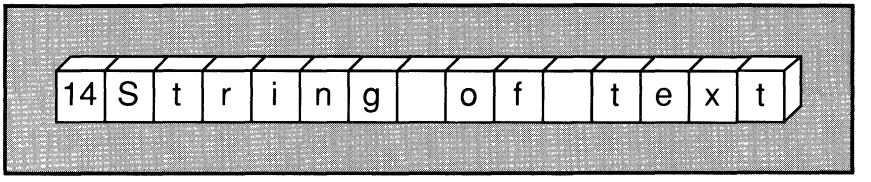


Figure 8.5 Pascal String Format

8.4.2 Passing BASIC Strings

When a BASIC string (such as A$) appears in an argument list, BASIC passes a string descriptor rather than the string data itself. The BASIC string descriptor is not compatible with the string formats of the other languages.

Warning

When you pass a string from BASIC to another language, the called routine should under no circumstances alter the length of the string. Other languages lack BASIC's string-space management routines. Therefore, altering the length of a BASIC string is liable to corrupt parts of the BASIC string space. Changes that do not affect length, however, are relatively safe.

The routine that receives the string must not call any BASIC routine. If it does, BASIC's string-space management routines may change the location of the string data without warning.

However, the **SADD** and **LEN** functions extract parts of the string descriptor. **SADD** extracts the address of the actual string data, and **LEN** extracts the length. The results of these functions can then be passed to other languages.

BASIC should pass the result of the **SADD** function by value. Bear in mind that the string's *address*, not the string itself, will be passed by value. This amounts to passing the string itself by reference. The BASIC

module passes the string address, and the other module receives the string address. The address returned by **SADD** is declared as type integer, but is actually equivalent to a C near pointer or Pascal **ADR** variable.

Pass LEN(A$) as you would normally pass a two-byte integer.

■ Passing BASIC Strings to C

Before attempting to pass a BASIC string to C, you may want to first append a null byte on the end, with an instruction such as:

```
A$ = A$ + CHR$(0)
```

The string now conforms to the C string format. Note that when used in a BASIC string expression the + indicates concatenation, or joining, of two strings.

There are two methods for passing a string from BASIC to C. The first method is to pass the string address and string length as separate arguments, using the **SADD** and **LEN** functions. If you are linking to a C library routine, this is the only workable method.

```
DECLARE SUB Test CDECL(BYVAL S%, BYVAL N%)
CALL Test(SADD(A$), LEN(A$))
.
.
.
void Test(s, n)
char near *s;
int  n;
{
```

In the example above, SADD(A$) returns the near address of the string data. This address must be passed by value, since it is equivalent to a pointer (even though treated by BASIC as an integer). Passing by reference would attempt to pass the *address* of the address, rather than the address itself.

C must receive a near pointer since only the near (offset) address is being passed by BASIC. Near pointers are the default pointer size in medium-model C.

The second method is to pass the string descriptor itself, with a call statement such as:

```
CALL Test2(A$)
```

In this case, the C function must declare a structure for the parameter, which has the appropriate fields (length and address) for a BASIC string descriptor. The C function should then expect to receive a pointer to a structure of this type.

■ Passing BASIC Strings to FORTRAN

FORTRAN variable-length strings (available in Version 4.0) cannot be used in a mixed-language interface.

Use the **SADD** function to pass the address of a BASIC string. The FORTRAN routine should declare a character variable of the same length (which is fixed).

```
DECLARE SUB Test(BYVAL S%)
A$="abcd"
CALL(SADD(A$))
.
.
.
C       FORTRAN SOURCE
C
        SUBROUTINE  TEST(STRINGA)
        CHARACTER*4 STRINGA [NEAR]
```

In the example above, SADD(A$) should be passed by value, since it is actually an address and not an integer. (Passing a string by reference is equivalent to passing the string *address* by value.) Note that CHARACTER*4 STRINGA [NEAR] declares a fixed-length parameter received by near reference.

FORTRAN must receive by near reference. The **NEAR** attribute makes this adjustment, since the FORTRAN default is to receive by far reference.

■ Passing BASIC Strings to Pascal

The same technique used to pass a string to FORTRAN can be used to pass a string to Pascal. However, the Pascal routine should declare the string as a **VAR** parameter, in order to receive the string by near reference. The Pascal code must declare the fixed-length type `string(4)` in a separate statement, then use the declared type in a **procedure** declaration.

```
DECLARE SUB Test(BYVAL S%)
A$="abcd"
CALL Test(SADD(A$))
.
.
.
type stype4=string(4);
procedure Test(VAR StringA:stype4);
```

8.4.3 Passing C Strings

When a C string appears in an argument list, C passes the address of the string. (A C string is just an array and so is passed by reference.) C can easily pass data to a fixed-length FORTRAN or Pascal string, or to BASIC in the form of a string descriptor.

■ Passing C Strings to BASIC

To pass a C string to BASIC, first allocate a string in C. Then create a structure identical to a BASIC string descriptor. Pass this structure by near reference, as in the example below:

```
char cstr[] = "ABC";
struct {
        char    *sd_addr;
        int     sd_len;
} str_des;
str_des.sd_addr = cstr;
str_des.sd_len = strlen(cstr);
bsub(&str_des);
```

Make sure that the string originates in C, not in BASIC. Otherwise, BASIC may attempt to move the string around in memory.

■ Passing C Strings to FORTRAN and Pascal

To pass strings to FORTRAN and Pascal, it is only necessary to make sure that the called routine receives the string by reference and allocates sufficient space. FORTRAN and Pascal should expect to receive fixed-length strings; to declare a fixed-length string parameter in Pascal you must first declare a type, as shown below.

■ Example

```
/* C code - calls Pascal and FORTRAN */
/* large memory model assumed */

char a[]="abcd";

Test1(a);                /* call to FORTRAN */
Test2(a);                /* call to Pascal */


{* Pascal *}
module Ptest1;
type stype4 : string(4);
procedure Test1(vars StringA : stype4)


C       FORTRAN
C
        SUBROUTINE TEST2(A)
        CHARACTER*4 A
```

However, C cannot pass variable-length strings to FORTRAN; the FORTRAN string data are not placed on the stack, but require special low-level variables found only in a FORTRAN program.

8.4.4 Passing FORTRAN Strings

Variable-length FORTRAN strings of type **CHARACTER*(*)** (available in Versions 4.0 and later) cannot be effectively passed to other languages. However, fixed-length strings can be passed without much difficulty; the principal limitation is that the FORTRAN **INTERFACE** must declare the length of the string in advance.

By default, FORTRAN passes strings by reference. However, if you apply the **C** or **PASCAL** attribute to a routine, then the default changes to passing by value. The actual string data do not include a delimiter, unless you use the C-string feature described below.

■ Passing FORTRAN Strings to BASIC

FORTRAN cannot directly pass strings to BASIC because BASIC expects to receive a string descriptor when passed a string. Yet there is an indirect method for passing FORTRAN strings to BASIC. First, allocate a fixed-length string in FORTRAN, declare an array of two-byte integers, and treat the array as a string descriptor. Next, assign the address of the

string to the first element (using the **LOC** function), and assign the length of the string to the second element. Finally, pass the integer array itself by reference. BASIC can receive and process this array just as it would a string descriptor.

■ Passing FORTRAN Strings to C

The C-string feature overrides the normal FORTRAN format and produces strings that can be effectively manipulated by C. When the C-string feature is used, a null byte is appended to the end of the string, and backslashes that appear in a literal-string text are treated as escapes.

You convert FORTRAN strings to C strings by simply typing **C** immediately after a string constant. Do not insert commas or any other intervening punctuation, only spaces. Note that the length of the string is increased by one because of the null byte that is appended. You need to allow for this when you declare string variables.

The following example passes the address of a string to C. The string is in the C format.

■ Example

```
        INTERFACE TO SUBROUTINE CONV [C] (S1)
        CHARACTER*5 S1 [REFERENCE]
        END
C
        CHARACTER*5 S1
        S1 = 'abcd' C
        CALL CONV (S1)
```

In the example above, note that an additional byte is allocated for `S1`, in consideration of the null byte added by the C-string conversion (done on the line above the call). Also note that the **REFERENCE** keyword was necessary because the **C** attribute in the first line changes the parameter-passing default to calling by value.

■ Passing FORTRAN Strings to Pascal

The FORTRAN and Pascal fixed-length string types are equivalent and therefore can be easily passed between FORTRAN and Pascal.

FORTRAN modules should only pass fixed-length strings to Pascal. The Pascal routines, in turn, should expect to receive fixed-length strings. To specify a fixed-length string parameter in Pascal, you first need to declare a type, as in the example below.

■ **Example**

```
C   FORTRAN SOURCE CODE
C
        INTERFACE TO SUBROUTINE PS (S1)
        CHARACTER*4 S1
        END
C
        S1 = 'wxyz'
        CALL PS (S1)
        END
```

```
{ Pascal module}

module Psmod;
type stype4 = string(4);
procedure ps (vars str1 : stype4);
```

8.4.5 Passing Pascal Strings

The Pascal data type **LSTRING** is not compatible with the formats used by the other languages. You can pass an **LSTRING** indirectly, however, by first assigning it to a **STRING** variable. Pascal supports such assignments by performing a conversion of the data.

Important

Pascal passes an additional, two-byte parameter that indicates string length whenever you pass a parameter of type **STRING** or of type **LSTRING**. To suppress the passing of this additional parameter, you first declare a fixed-length type, as shown in the example in the section, "Passing Pascal Strings to C," below.

■ **Passing Pascal Strings to BASIC**

To pass a Pascal string to BASIC, first allocate a string in Pascal. Next, create a record identical to a BASIC string descriptor. Initialize this record with the string address and length, and then pass the record by near reference. Make sure that the string originates in Pascal, not in BASIC; otherwise, BASIC may attempt to move the string data around in memory.

■ **Passing Pascal Strings to C**

To pass a string to C, first append a null character (numerical 0, ASCII NUL) to the end of the string by using the concatenation operator (*).

Then pass the string to C by reference (by declaring the string argument as **CONST**, **CONSTS**, **VAR**, or **VARS**). Remember to first declare the fixed-length string type.

■ Example

```
program Passtr(input, output);
type
        stype6 = string(6);
var
        str : stype6;
procedure Passtoc (var s1 : stype6) [C]; extern;
begin
        str := 'abcde' * chr(0);
        Passtoc(str);
```

You can achieve more flexibility in passing Pascal strings by declaring a value parameter of type **ADRMEM** or **ADSMEM** and then passing the address of the argument. For instance, the example above could be implemented by first declaring the parameter with the statement,

```
procedure Passtoc (s1adr : ADRMEM) [C]; extern;
```

Then you could make the call with

```
        Passtoc(ADR str);
```

With this method, you can pass strings of different lengths to the procedure `Passtoc`.

■ Passing Pascal Strings to FORTRAN

The Pascal **STRING** and the FORTRAN **CHARACTER*n** types are equivalent. Therefore Pascal fixed-length string variables can be freely passed to FORTRAN. Usually, you will find it most efficient to pass strings by reference (by declaring the string argument as **VAR** or **VARS**). Remember to first declare the fixed-length type before using it.

```
program Passtr(input, output)
type
        stype6 = string(6);
var
        str : stype6;
procedure PasstoF (var s1 : stype6); extern;
begin
        PasstoF(str);
```

As explained previously, you can use **ADRMEM** and **ADSMEM** to achieve more flexibility in passing strings from Pascal.

CHAPTER 9

SPECIAL DATA TYPES

This chapter considers special types of data that are either structured (i.e., contain more than one field) or are accessed externally.

9.1 Arrays

When you program in only one language, arrays do not present special problems; the language is consistent in its handling of arrays. When you program with more than one language, however, you need to be aware of two special problems that may arise with arrays:

1. Arrays are implemented differently in BASIC, so that you must take special precautions when you pass an array from BASIC to another language (including assembly).
2. Arrays are declared and indexed differently in each language.

This section considers each of these problems in turn.

Note

As explained in Chapter 7, arrays cannot be passed by value in C, unless declared within a structure. However, it is usually most efficient to pass arrays by reference.

9.1.1 Passing Arrays from BASIC

Most Microsoft languages permit you to reference arrays directly. In C, for example, an address name is equivalent to the address of the first element. FORTRAN and Pascal are similar. This simple implementation is possible because the location of data for an array never changes.

BASIC uses an *array descriptor*, however, which is similar in some respects to a BASIC string descriptor. The array descriptor is necessary because BASIC may shift the location of array data in memory; BASIC handles memory allocation for arrays dynamically.

C, FORTRAN, and Pascal do not have any equivalent of the BASIC array descriptor. More importantly, they lack access to BASIC's space management routines for arrays. Therefore, you may safely pass arrays from BASIC only if you follow three rules:

1. Pass the array's address by applying the **VARPTR** function to the first element of the array and passing the result by value. To pass the far address of the array, apply both the **VARPTR** and

VARSEG functions and pass each result by value. The receiving language gets the address of the first element and considers it to be the address of the entire array. It can then access the array with its normal array-indexing syntax. The example below illustrates how this works.

2. The routine that receives the array must not, under any circumstances, make a call back to BASIC. If it does, then the location of the array data may change, and the address that was passed to the routine will become meaningless.
3. BASIC may pass any member of an array by value. With this method, the above precautions do not apply.

The following example demonstrates how a BASIC array can be passed to FORTRAN.

■ Example

```
REM   BASIC SOURCE FILE
OPTION BASE 1
DEFINT A-Z
DIM A(20)
DECLARE SUB ArrFix(BYVAL Addr AS INTEGER)
.
.
.
CALL ArrFix(VARPTR(A(1)))
PRINT A(1)
END


C   FORTRAN SOURCE FILE
C
        SUBROUTINE ARRFIX (ARR)
        INTEGER*2 ARR [NEAR]  (20)
        ARR(1) = 5
        END
```

In the example above, BASIC considers that the argument passed is the near address of an array element. FORTRAN considers it to be the near address of the array itself. Both languages are correct. You can use essentially the same method for passing BASIC arrays to Pascal or C.

The parameter was declared BYVAL Addr AS INTEGER because a near (two-byte) address needed to be passed. If you wanted to pass a far (four-byte) address, then the proper code would be:

```
DECLARE SUB ArrFix (BYVAL SegAddr AS INTEGER, BYVAL Addr AS INTEGER)
CALL ArrFix (VARSEG(A(0)), VARPTR(A(0)))
```

The first field is the segment returned by **VARSEG**. If you use **CDECL** then be sure to pass the offset address before the segment address, because

CDECL causes parameters to be passed in reverse order:

```
DECLARE SUB ArrFix CDECL (BYVAL Addr AS INTEGER, BYVAL SegAddr AS INTEGER)
CALL ArrFix(VARPTR(A((0)), VARSEG(A(0)))
```

Note

You can apply **LBOUND** and **UBOUND** to a BASIC array, to determine lower and upper bounds, and then pass the results to another routine. This way, the size of the array does not need to be determined in advance. See the *Microsoft BASIC Language Reference* for more information on **LBOUND** and **UBOUND**.

9.1.2 Array Declaration and Indexing

Each language varies somewhat in the way that arrays are declared and indexed. Array indexing is purely a source-level consideration and involves no transformation of data. There are two differences in the way that elements are indexed by each language:

1. Lower bounds.

   By default, FORTRAN indexes the first element of an array as 1. BASIC and C index it as 0. Pascal lets the programmer begin indexing at any integer value. Recent versions of BASIC and FORTRAN also give the user the option of specifying lower bounds at any integer values.

2. Row-major order vs. column-major order.

   This issue only affects arrays with more than one dimension. With row-major order (used by C and Pascal) the leftmost dimension changes the fastest. With column-major order (used by FORTRAN, and BASIC by default), the rightmost dimension changes the fastest. Thus, in Pascal the first four elements of array `X(3,3)` are:

   ```
   X[1,1]   X[1,2]   X[1,3]   X[2,1]
   ```

   In FORTRAN, the four elements are:

   ```
   X(1,1)   X(2,1)   X(3,1)   X(1,2)
   ```

The example above assumes that both the Pascal and FORTRAN arrays use lower bounds of 1. Table 9.1 shows equivalences for array declarations in each language. In this table, *r* is the number of elements of the row dimension (which changes most slowly), and *c* is the number of elements of the column dimension (which changes most quickly).

Table 9.1
Equivalent Array Declarations

Language	Array Declaration	Notes
BASIC	**DIM** *x(c-1,r-1)*	with default lower bounds of 0
C	*type x[r][c]*	when passed by reference
	struct { *type x[r][c];*} *x*	when passed by value
FORTRAN	*type x(c,r)*	with default lower bounds of 1
Pascal	*x*:**array**[*a..a+r-1,b..b+c-1*] **of** *type*	. . .

The declarations above extend to any number of dimensions that you may use. For example, the C declaration

```
int     arr1 [2] [10] [15] [20]
```

is equivalent to the FORTRAN declaration

```
INTEGER*2 ARR1(20, 15, 10, 2)
```

■ Example

The following references all refer to the same place in memory for an array:

```
arr1[2][8]              (in C)
Arr1[3,9]               (in Pascal, assuming lower bounds of 1)
ARR1(9,3)               (in FORTRAN, assuming lower bounds of 1)
ARR1(8,2)               (in BASIC, assuming lower bounds of 0)
```

When you use BASIC with the **BC** command line, you can select the **/R** compile option, which specifies that row-major order is to be used, rather column-major order.

Note

The constants used in a C array declaration represent dimensions, not upper bounds as they do in other languages. Therefore, the last element in the C array declared as `int arr[5][5]` is not `arr[5][5]`, but `arr[4][4]`.

9.2 Structures, Records, and User-defined Types

The C **struct** type, the BASIC user-defined type, and the Pascal **record** type are equivalent. Therefore, these data types can be passed between C, Pascal, and BASIC.

However, these types may be affected by the storage method. By default, C and Pascal use word alignment (unpacked storage) for all data except byte-sized objects and arrays of byte-sized objects. This storage method specifies that occasional bytes may be added as padding, so that word and double-word objects start on an even boundary. (In addition, all nested structures and records start on a word boundary.) The following illustration shows the the contrast between packed and unpacked storage:

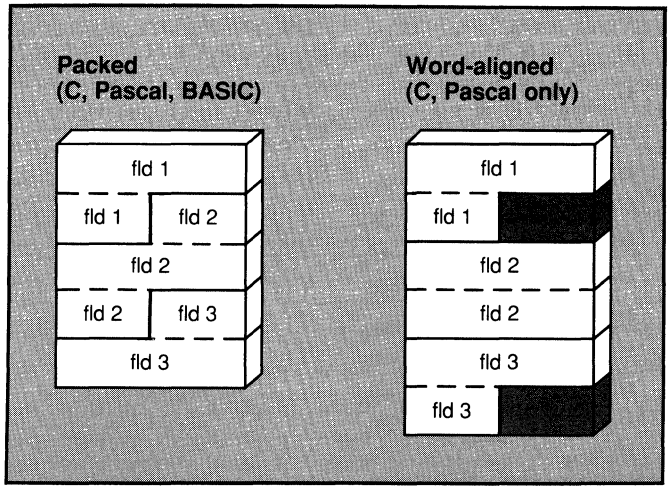


Figure 9.1 Structure and Record Storage

If a structure or record is being passed between them, it is important that a calling routine and the called routine agree on storage method. Otherwise, data will not be properly received. The simplest method for ensuring compatibility between all three languages is simply to turn on packing for C and Pascal modules. The packed storage method may sacrifice some speed, but it has the advantage of creating smaller executable files.

9.3 External Data

You can always share data between two languages by passing parameters. In the case of local variables and all BASIC variables, passing parameters is the only convenient way to share data.

However, C, FORTRAN, and Pascal routines can access data directly that are external. The term "external" refers to data that are both static and public; in other words, the data are stored in a set place in memory (static, unlike dynamic or local data, which are allocated on the stack), and the data have been made publicly available to other modules. Compilers make a data object (variable, structure or array) available by placing its name, along with size and type information, into the object file.

External data (data that can be directly accessed by any other module) can be defined in a C, FORTRAN, Pascal, or assembly module. Note that a data *definition* is distinct from an external *declaration.* A definition causes a compiler to create a data object; an external declaration informs a compiler that the object is to be found in another module.

There are three requirements for programs that share external data between languages:

1. One of the modules must define the static data.

   You can define a static data object in a C or FORTRAN module by defining a data object outside all functions and subroutines. (Do not use the **static** keyword in C with a data object you wish to be public.)

2. The other modules that will access the data must declare the data as external.

   In C, you can declare data as external by using an **extern** declaration, similar to the **extern** declaration for functions. In FORTRAN and Pascal, you can declare data as external by adding the **EXTERN** attribute to the data declaration.

3. Resolve naming-convention differences.

In C, you can adopt the BASIC/FORTRAN/Pascal naming convention by applying **fortran** or **pascal** to the data declaration. In FORTRAN and Pascal, you can adopt the C naming convention by applying the **C** attribute to the data declaration.

The examples below help illustrate the general language features of external data just described.

■ Examples

```
/* C source code */

int       thing1;            /* Thing1 is public and static */
extern    int thing2;        /* Thing2 is defined in another module */
static    int thing3;        /* Thing3 is static, but not public */

ctest()
{
.
.
.
C   FORTRAN SOURCE CODE
C
          INTEGER*2 THING1 [C, EXTERN]
          INTEGER*2 THING2 [C]
C
C   THING1 DEFINED IN ANOTHER MODULE, USING C CONVENTION (_thing1)
C   THING2 DEFINED HERE, USING C CONVENTION (_thing2)
.
.
.
{ Pascal source code }

module Ptest;
 procedure Test;
    var
       thing1 [C, EXTERN] : integer; { Both vars defined elsewhere }
       thing2 [C, EXTERN] : integer; {  and use C naming convention }
.
.
.
```

In the examples above, the variables `thing1` and `thing2` are defined and declared with the C naming convention so that they will be placed into each object file as `_thing1` and `_thing2`. However, you can just as easily specify the BASIC/FORTRAN/Pascal naming convention, by using the following C statements:

```
int      fortran  thing1;
extern   int  fortran thing2;
```

The **C** attribute can then be dropped from the FORTRAN and Pascal source code. Each object file will contain the names `THING1` and `THING2`.

9.4 Pointers and Address Variables

Rather than passing data directly, you may want to pass the *address* of a piece of data. Passing the address amounts to passing the data itself by reference. In some cases, such as BASIC arrays (see Section 9.1.1), passing an address is the only way to share particular kinds of data between two languages.

The Pascal **ADR** and **ADS** types are equivalent to near and far pointers, respectively, in C. You can pass **ADR** and **ADS** variables as **ADRMEM** or **ADSMEM**. BASIC and FORTRAN do not have formal address types. However, they do provide ways for storing and passing addresses.

BASIC programs can access a variable's segment address with **VARSEG** and its offset address with **VARPTR**. The values returned by these intrinsic functions should then be passed or stored as ordinary integer variables. If you pass them to another language, pass by value. Otherwise you will be attempting to pass the *address* of the address, rather than the address itself.

To pass a near address, pass only the offset; if you need to pass a far address, you may need to pass the segment and offset separately. Pass the segment address first, unless **CDECL** is in effect.

FORTRAN programs can determine near and far addresses with the **LOC** and **LOCFAR** functions. Store the result as **INTEGER*2** (with the **LOC** function) or as **INTEGER*4** (with the **LOCFAR** function).

As with BASIC, if you pass the result of **LOC** or **LOCFAR** to another language, be sure to pass by value.

9.5 Common Blocks

You can pass individual members of a FORTRAN or BASIC common block in an argument list, just as you can with any data. However, you can also give a different language module access to the entire common block at once.

Pascal and C modules can reference the items of a common block by first declaring a structure or record, with fields that correspond to the common block variables. (For an example, see the next section.) BASIC modules can also employ a user-defined type to access the fields of a FORTRAN common block.

Having defined a structure, record, or user-defined type with the appropriate fields, the Pascal or C module must then connect with the common block itself. The next two sections each present a method for gaining access to common blocks.

9.5.1 Passing the Address of the Common Block

To pass the address of a common block, simply pass the address of the first variable in the block. (In other words, pass the first variable by reference.) The receiving C or Pascal module should expect to receive a structure (or record) by reference.

■ Example

In the example below, the C function `initcb` receives the address of the variable N, which it considers to be a pointer to a structure with three fields:

```
C   FORTRAN SOURCE CODE
C
        COMMON /CBLOCK/N,X,Y
        INTEGER*2 N
        REAL*8 X,Y
.
.
.
        CALL INITCB(N)
.
.
.
/* C source code */

struct block_type {
        int     n;
        double  x;
        double  y;
};

initcb(block_hed)
struct block_type *block_hed;
{
        block_hed->n = 1;
        block_hed->x = 10.0;
        block_hed->y = 20.0;
}
```

9.5.2 Accessing Common Blocks Directly

You can access FORTRAN common blocks directly by defining a structure (or record in Pascal) with the appropriate fields and then using the methods described in Section 9.3, "External Data."

■ Example

In the example below, `cblock` is declared as an external structure. You can reference the individual fields of `cblock` which will correspond to those of the common block CBLOCK in the FORTRAN source file.

```
struct block_type {
        int     n;
        double  x;
        double  y;
};

extern struct block_type fortran cblock;
```

9.6 Using a Varying Number of Parameters

Some C functions, most notably **printf**, can be called with a different number of arguments each time. To call such a function from another language, you need to suppress the type-checking that normally forces a call to be made with a fixed number of parameters. In BASIC, you can remove this type checking by omitting from the **DECLARE** statement a parameter list, as explained in Section 2.2, "Alternative BASIC Interfaces." In FORTRAN or Pascal, you can call routines with a variable number of parameters by including the **VARYING** attribute in your interface to the routine, along with the **C** attribute. You must use the **C** attribute because a variable number of parameters is only feasible with the C calling convention.

The **VARYING** attribute prevents FORTRAN or Pascal from enforcing a matching number of parameters. Each time you call the routine, you will be able to pass more or fewer parameters than are declared in the interface to the routine. However, each actual parameter that you pass will be type-checked against whatever formal parameters you may have declared in the interface. FORTRAN or Pascal will compare the type of the first actual parameter to the first formal parameter (if any), the second actual parameter to the second formal parameter, and so on.

Because the number of parameters is not fixed, the routine you call should have some mechanism for determining how many parameters to expect. Often this information is indicated by the first parameter. For example, the C function **printf** scans the format string passed as the first parameter. The number of fields in the format string determines how many additional parameters the function should expect.

The example below demonstrates the use of the **VARYING** attribute to call **printf** directly from Pascal (the program needs to be compiled and linked to the C large memory model so that **printf** is linked in).

■ Example

```
program Test (input, output);
type
        stype30 : string(30);
var
        str1 : string(30);
        str2 : string(10);
        n    : integer;
procedure printf (vars s1 : stype30) [C, VARYING]; extern;
begin
        str1 = 'This is %s string, number %d.' * chr(0);
        str2 = 'formatted' * chr(0);
        n = 1;
        printf(str1, str2, n);
end.
```

In Pascal, you can write the interface to **printf** so that the format string can be of varying lengths, by using the **ADRMEM** feature. See Section 8.4.5, "Passing Pascal Strings," for more information.

MIXED-LANGUAGE PROGRAMMING GUIDE INDEX

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®
Making it all make sense™

1287 Part No. 01508