CHAPTER 12

Mixed-Language Programming

Mixed-language programming allows you to combine the unique strengths of Microsoft Basic, C, C++, and FORTRAN with your assembly-language routines. Any one of these languages can call MASM routines, and you can call any of these languages from within your assembly-language programs. This makes virtually all the routines from high-level–language libraries available to a mixed-language program.

MASM 6.1 provides mixed-language features similar to those in high-level languages. For example, you can use the **INVOKE** directive to call high-level-language procedures, and the assembler handles the argument-passing details for you. You can also use H2INC to translate C header files to MASM include files, as explained in Chapter 20 of *Environment and Tools*.

The mixed-language features of MASM 6.1 do not make older methods of defining mixed-language interfaces obsolete. In most cases, mixed-language programs written with earlier versions of MASM will assemble and link correctly under MASM 6.1. (For more information, see Appendix A.)

This chapter explains how to write assembly routines that can be called from high-level–language modules and how to call high-level language routines from MASM. You should already understand the languages you want to combine and should know how to write, compile, and link multiple-module programs with these languages.

This chapter covers only assembly-language interface with C, C++, Basic, and FORTRAN; it does not cover mixed-language programming between high-level languages. The focus here is the Microsoft versions of C, C++, Basic, and FORTRAN, but the same principles apply to other languages and compilers. Many of the techniques used in this chapter are explained in the material in Chapter 7 on writing procedures in assembly language, and in Chapter 8 on multiple-module programming.

The first section of this chapter discusses naming and calling conventions. The next section, “Writing an Assembly Procedure for a Mixed-Language Program,” provides a template for writing an assembly-language procedure that can be

called from another module written in a high-level language. This represents the essence of mixed-language programming. Assembly language is often used for creating fast secondary routines in a large program written in a high-level language.

The third section describes specific conventions for linking assembly-language procedures with modules in C, C++, Basic, and FORTRAN. These language-specific sections also provide details on how the language manages various data structures so that your MASM programs are compatible with the data from the high-level language.

Naming and Calling Conventions

Each language has its own set of conventions, which fall into two categories:

- The "naming convention" specifies how or if the compiler or assembler alters the name of an identifier before placing it into an object file.
- The "calling convention" determines how a language implements a call to a procedure and how the procedure returns to the caller.

MASM supports several different conventions. The assembler uses C convention when you specify a language type (*langtype*) of **C**, and Pascal convention for language types **PASCAL**, **BASIC**, or **FORTRAN**. To the assembler, the keywords **BASIC**, **PASCAL**, and **FORTRAN** are synonymous. MASM also supports the **SYSCALL** and **STDCALL** conventions, which mix elements of the C and Pascal conventions.

MASM gives you several ways to set the naming and calling conventions in your assembly-language program. Using **.MODEL** with a *langtype* sets the default for the module. This can also be done with the **OPTION** directive. This is equivalent to the /Gc or /Gd option from the command line. Procedure prototypes and declarations can specify a *langtype* to override the default.

When you write mixed-language routines, the easiest way to ensure convention compatibility is to adopt the conventions of the called procedure's language. However, Microsoft languages can change the naming and calling conventions for different procedures. If your program must call a procedure that uses an argument-passing method different from that of the default language, prototype the procedure first with the desired language type. This tells the assembler to override the conventions of the default language and assume the proper conventions for the prototyped procedure. "The MASM/High-Level–Language Interface" section in this chapter explains how to change the default conventions. The following sections provide more detail on the information summarized in Table 12.1.

Table 12.1 Naming and Calling Conventions

Convention	C	SYSCALL	STDCALL	BASIC	FORTRAN	PASCAL
Leading underscore	X		X			
Capitalize all				X	X	X
Arguments pushed left to right				X	X	X
Arguments pushed right to left	X	X	X			
Caller stack cleanup	X	X	*			
:VARARG allowed	X	X	X			

* The STDCALL language type uses caller stack cleanup if the :VARARG parameter is used. Otherwise, the called routine must clean up the stack.

Naming Conventions

"Naming convention" refers to the way a compiler or assembler stores the names of identifiers. The first two rows of Table 12.1 show how each language type affects symbol names. **SYSCALL** leaves symbol names as they appear in the source code, but **C** and **STDCALL** add an underscore prefix. **PASCAL**, **BASIC**, and **FORTRAN** change symbols to all uppercase.

The following list describes how these naming conventions affect a variable called `Big Time` in your source code:

Langtype Specified	Characteristics
SYSCALL	Leaves the name unmodified. The linker sees the variable as `Big Time`.
C, **STDCALL**	The assembler (or compiler) adds a leading underscore to the name, but does not change case. The linker sees the variable as `_Big Time`.
PASCAL, **FORTRAN**, **BASIC**	Converts all names to uppercase. The linker sees the variable as `Big Time`.

The C Calling Convention

Specify the C language type for assembly-language procedures called from programs that assume the C calling convention. Note that such programs are not necessarily written in C, since other languages can mimic C conventions.

Argument Passing

With the C calling convention, the caller pushes arguments from right to left as they appear in the caller's argument list. The called procedure returns without removing the arguments from the stack. It is the caller's responsibility to clean the stack after the call, either by popping the arguments or by adding an appropriate value to the stack pointer SP.

Register Preservation

The called routine must return with the original values in BP, SI, DI, DS, and SS. It must also preserve the direction flag.

Varying Number of Arguments

The additional overhead of cleaning the stack after each call has compensations. It frees the caller from having to pass a set number of arguments to the called procedure each time. Because the first argument in the list is always the last one pushed, it is always on the top of the stack. Thus, it has the same address relative to the frame pointer, regardless of how many arguments were actually passed.

For example, consider the C library function **printf**, which accepts different numbers of arguments. A C program calls the function like this:

```
printf( "Numbers:  %f  %f  %.2f\n", n1, n2, n3 );
printf( "Also:     %f", n4 );
```

The first line passes four arguments (including the string in quotes) and the second line passes only two arguments. Notice that **printf** has no reliable way of determining how many arguments the caller has pushed. Therefore, the function returns without adjusting the stack. The C calling convention requires the caller to take responsibility for removing the arguments from the stack, since only the caller knows how many arguments it passed.

Use **INVOKE** to call a C-callable function from your assembly-language program, since **INVOKE** automatically generates the necessary stack-cleaning code after the call. You must also prototype the function with the **VARARG** keyword if appropriate, as explained in "Procedures," Chapter 7. Similarly, when you write a C-callable procedure that accepts a varying number of arguments, include **VARARG** in the procedure's **PROC** statement.

The Pascal Calling Convention

By default, the *langtype* for **FORTRAN**, **BASIC**, and **PASCAL** selects the Pascal calling convention. This convention pushes arguments left to right so that the last argument is lowest on the stack, and it requires that the called routine remove arguments from the stack.

Argument Passing

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

Register Preservation

A routine that uses the Pascal calling convention must preserve SI, DI, BP, DS, and SS. For 32-bit code, the EBX, ES, FS, and GS registers must be preserved as well as EBP, ESI, and EDI. The direction flag is also cleared upon entry and must be preserved.

Varying Number of Arguments

Passing a variable number of arguments is not possible with the Pascal calling convention.

The STDCALL and SYSCALL Calling Conventions

A **STDCALL** procedure adopts the C name and calling conventions when prototyped with the **VARARG** keyword. Refer to the section "Declaring Parameters with the PROC Directive" in Chapter 7. Without **VARARG**, the procedure uses the C naming and Pascal calling conventions. **STDCALL** provides compatibility with 32-bit versions of Microsoft compilers.

As Table 12.1 shows, **SYSCALL** is identical to the C calling convention, but does not add an underscore prefix to symbols.

Argument Passing

Argument passing order for both **STDCALL** and **SYSCALL** is the same as the C calling convention. The caller pushes the arguments from right to left and must remove the parameters from the stack after the call. However, **STDCALL** requires the called procedure to clean the stack if the procedure does not accept a variable number of arguments.

Register Preservation

Both conventions require the called procedure to preserve the registers BP, SI, DI, DS, and SS. Under **STDCALL**, the direction flag is clear on entry and must be returned clear.

Varying Number of Arguments

SYSCALL allows a variable number of arguments in the same way as the C calling convention. **STDCALL** also mimics the C convention when **VARARG** appears in the called procedure's declaration or definition. It allows a varying number of arguments and requires the caller to clean the stack. If not declared

or defined with **VARARG**, the called procedure does not accept a variable argument list and must clean the stack before it returns.

Writing an Assembly Procedure For a Mixed-Language Program

MASM 6.1 simplifies the coding required for linking MASM routines to high-level–language routines. You can use the **PROTO** directive to write procedure prototypes, and the **INVOKE** directive to call external routines. MASM simplifies procedure-related tasks in the following ways:

- The **PROTO** directive improves error checking on argument types.
- **INVOKE** pushes arguments onto the stack and converts argument types to types expected when possible. These arguments can be referenced by their parameter label, rather than as offsets of the stack pointer.
- The **LOCAL** directive following the **PROC** statement saves places on the stack for local variables. These variables can also be referenced by name, rather than as offsets of the stack pointer.
- **PROC** sets up the appropriate stack frame according to the processor mode.
- The **USES** keyword preserves registers given as arguments.
- The C calling conventions specified in the **PROC** syntax allow for a variable number of arguments to be passed to the procedure.
- The **RET** keyword adjusts the stack upward by the number of bytes in the argument list, removes local variables from the stack, and pops saved registers.
- The **PROC** statement lists parameter names and types. The parameters can be referenced by name inside the procedure.

The complete syntax and parameter descriptions for these procedure directives are explained in "Procedures" in Chapter 7. This section provides a template that you can use for writing a MASM routine to be called from a high-level language.

The template looks like this:

Label **PROC** [[*distance langtype visibility <prologueargs>* **USES** *reglist parmlist*]]
    LOCAL *varlist*
    .
    .
    .

RET
Label **ENDP**

Replace the italicized words with appropriate keywords, registers, or variables as defined by the syntax in "Declaring Parameters with the **PROC** Directive" in Chapter 7.

The *distance* (**NEAR** or **FAR**) and *visibility* (**PUBLIC**, **PRIVATE**, or **EXPORT**) that you give in the procedure declaration override the current defaults. In some languages, the model can also be specified with command-line options.

The *langtype* determines the calling convention for accessing arguments and restoring the stack. For information on calling conventions, see "Naming and Calling Conventions" earlier in this chapter.

The types for the parameters listed in the *parmlist* must be given. Also, if any of the parameters are pointers, the assembler does not generate code to get the value of the pointer references. You must write this code yourself. An example of how to write such code is provided in "Declaring Parameters with the PROC Directive" in Chapter 7.

If you need to code your own stack-frame setup manually, or if you do not want the assembler to generate the standard stack setup and cleanup, see "Passing Arguments on the Stack" and "User-Defined Prologue and Epilogue Code" in Chapter 7.

The MASM/High-Level–Language Interface

Since high-level–language programs require initialization, you must write the main routine of a mixed-language program in the high-level language, or link with the startup code supplied by the high-level–language compiler. This gives the assembly code access to high-level routines or library functions. The next section explains how to link an assembly-language program with C-language startup code.

For procedures with prototypes, **INVOKE** makes calls from MASM to high-level–
language programs, much like procedure or function calls in the high-level language. **INVOKE** calls procedures and generates the code to push arguments in the order specified by the procedure's calling convention, and to remove arguments from the stack at the end of the procedure.

INVOKE can also do type checking and data conversion for the argument types so that the procedure receives compatible data. For explanations of how to write procedure prototypes and several examples of procedure declarations and the corresponding prototypes, see "Declaring Procedure Prototypes" in Chapter 7.

For programs that mix assembly language and C, the H2INC utility makes it easy to write prototypes and data declarations for the C procedures you want to call from MASM. H2INC translates the C prototypes and declarations into the corresponding MASM prototypes and declarations, which **INVOKE** can use to call the procedure. The use of H2INC is explained in Chapter 20 in *Environment and Tools*.

Mixed-language programming also allows the main program or a routine to use external data—data defined in the other module. External data is the data that is stored in a set place in memory (unlike dynamic and local data, which is allocated on the stack and heap) and is visible to other modules.

External data is shared by all routines. One of the modules must define the static data, which causes the compiler to allocate storage for the data. The other modules that access the data must declare the data as external.

Argument Passing

Each language has its own convention for how an argument is actually passed. If the argument-passing conventions of your routines do not agree, then a called routine receives bad data. Microsoft languages support three different methods for passing an argument:

- Near reference. Passes a variable's near (offset) address, expressed as an offset from the default data segment. This method gives the called routine direct access to the variable itself. Any change the routine makes to the parameter is reflected in the calling routine.
- Far reference. Passes a variable's far (segmented) address. Though slower than passing a near reference, this method is necessary for passing data that lies outside the default data segment. (This is not an issue in Basic unless you have specifically requested far memory.)
- Value. Passes only a copy of the variable, not its address. With this method, the called routine gets a copy of the argument on the stack, but has no access to the original variable. The copy is discarded when the routine returns, and the variable retains its original value.

When you pass arguments between routines written in different languages, you must ensure that the caller and the called routine use the same conventions for passing and receiving arguments. In most cases, you should check the argument-passing defaults used by each language and make any necessary adjustments. Most languages have features that allow you to change argument-passing methods.

Register Preservation

A procedure called from any high-level language should preserve the direction flag and the values of BP, SI, DI, SS, and DS. Routines called from MASM must not alter SI, DI, SS, DS, or BP.

Pushing Addresses

Microsoft high-level languages push segment addresses before offsets. This lets the called routine use the **LES** and **LDS** instructions to read far addresses from the stack. Furthermore, each word of an argument is placed on the stack in order of significance. Thus, the high word of a long integer is pushed first, followed by the low word.

Array Storage

Most high-level-language compilers store arrays in row-major order. This means that all elements of a row are stored consecutively. The first five elements of an array with four rows and three columns are stored in row-major order as

```
A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2]
```

In column-major order, the column elements are stored consecutively. For example, this same array would be stored in column-major order as

```
A[1, 1], A[2, 1], A[3, 1], A[4, 1], A[1, 2], A[2, 2]
```

The C/MASM Interface

This section summarizes the characteristics of the interface between MASM and Microsoft C and QuickC compilers. With the default naming and calling convention, the assembler (or compiler) pushes arguments right to left and adds a leading underscore to routine names.

Compatible Data Types

This list shows the 16-bit C data types and equivalent data types in MASM 6.1. For 32-bit C compilers, **int** and **unsigned int** are equivalent to the MASM types **SDWORD** and **DWORD**, respectively.

C Type	Equivalent MASM Type
unsigned char	**BYTE**
char	**SBYTE**
unsigned short, **unsigned int**	**WORD**
int, **short**	**SWORD**
unsigned long	**DWORD**
long	**SDWORD**

float	**REAL4**
double	**REAL8**
long double	**REAL10**

Naming Restrictions

C is case-sensitive and does not convert names to uppercase. Since C normally links with the /NOI command-line option, you should assemble MASM modules with the /Cx or /Cp option to prevent the assembler from converting names to uppercase.

Argument-Passing Defaults

C always passes arrays by reference and all other variables (including structures) by value. C programs in tiny, small, and medium model pass near addresses for arrays, unless another distance is specified. Compact-, large-, and huge-model programs pass far addresses by default. To pass by reference a variable type other than array, use the C-language address-of operator (**&**).

If you need to pass an array by value, declare the array as a structure member and pass a copy of the entire structure. However, this practice is rarely necessary and usually impractical except for very small arrays, since it can make substantial demands on stack space. If your program must maintain an array through a procedure call, create a temporary copy of the array in heap and provide the copy to the procedure by reference.

Changing the Calling Convention

Put **_pascal** or **_fortran** in the C function declaration to specify the Pascal calling convention.

Array Storage

Array declarations give the number of elements. `A1[a][b]` declares a two-dimensional array in C with `a` rows and `b` columns. By default, the array's lower bound is zero. Arrays are stored by the compiler in row-major order. By default, passing arrays from C passes a pointer to the first element of the array.

String Format

C stores strings as arrays of bytes and uses a null character as the end-of-string delimiter. For example, consider the string declared as follows:

```
char msg[] = "string of text"
```

The string occupies 15 bytes of memory as:

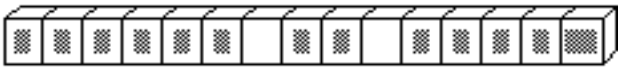

Figure 12.1 C String Format

Since `msg` is an array of characters, it is passed by reference.

External Data

In C, the **extern** keyword tells the compiler that the data or function is external. You can define a static data object in a C module by defining a data object outside all functions and subroutines. Do not use the **static** keyword in C with a data object that you want to be public.

Structure Alignment

By default, C uses word alignment (unpacked storage) for all data objects longer than 1 byte. This storage method specifies that occasional bytes may be added as padding, so that word and doubleword objects start on an even boundary. In addition, all nested structures and records start on a word boundary. MASM aligns on byte boundaries by default.

When converting .H files with H2INC, you can use the /Zp command-line option to specify structure alignment. If you do not specify the /Zp option, H2INC uses word-alignment. Without H2INC, set the alignment to 2 when declaring the MASM structure, compile the C module with /Zp1, or assemble the MASM module with /Zp2.

Compiling and Linking

Use the same memory model for both C and MASM.

Returning Values

The assembler returns simple data types in registers. Table 12.2 shows the register conventions for returning simple data types to a C program.

Table 12.2 Register Conventions for Simple Return Values

Data Type	Registers
char	AL
short, **near**, **int** (16-bit)	AX
short, **near**, **int** (32-bit)	EAX
long, **far** (16-bit)	High-order portion (or segment address) in DX; low-order portion (or offset address) in AX
long, **far** (32-bit)	High-order portion (or segment address) in EDX; low-order portion (or offset address) in EAX

Procedures using the C calling convention and returning type **float** or type **double** store their return values into static variables. In multi-threaded programs, this could mean that the return value may be overwritten. You can avoid this by using the Pascal calling convention for multi-threaded programs so **float** or **double** values are passed on the stack.

Structures less than 4 bytes long are returned in DX:AX. To return a longer structure from a procedure that uses the C calling convention, you must copy the structure to a global variable and then return a pointer to that variable in the AX register (DX:AX, if you compiled in compact, large, or huge model or if the variable is declared as a far pointer).

Structures, Records, and User-Defined Data Types

You can pass structures, records, and user-defined types as arguments by value or by reference.

Writing Procedure Prototypes

The H2INC utility simplifies the task of writing prototypes for the C functions you want to call from MASM. The C prototype converted by H2INC into a MASM prototype allows **INVOKE** to correctly call the C function. Here are some examples of C functions and the MASM prototypes created with H2INC.

```
/* Function Prototype Declarations to Convert with H2INC */

long checktypes (
        char *name,
        unsigned char a,
        int b,
        float d,
        unsigned int *num );

my_func (float fNum, unsigned int x);

extern my_func1 (char *argv[]);

struct videoconfig _far * _far pascal my_func2 (int, scri );
```

For these C prototypes, H2INC generates this code:

```
@proto_0        TYPEDEF         PROTO C :PTR SBYTE, :BYTE,
                                        :SWORD, :REAL4, :PTR WORD
checktypes      PROTO           @proto_0

@proto_1        TYPEDEF         PROTO C :REAL4, :WORD
my_func         PROTO           @proto_1

@proto_2        TYPEDEF         PROTO C :PTR PTR SBYTE
my_func1        PROTO           @proto_2

@proto_3        TYPEDEF         PROTO FAR PASCAL :SWORD, :scri
my_func2        PROTO           @proto_3
```

Example

As shown in the following short example, the main module (written in C) calls an assembly routine, `Power2`.

```
#include <stdio.h>

extern int Power2( int factor, int power );

void main()
{
    printf( "3 times 2 to the power of 5 is %d\n", Power2( 3, 5 ) );
}
```

Figure 12.2 shows how functions that observe the C calling convention use the stack frame.

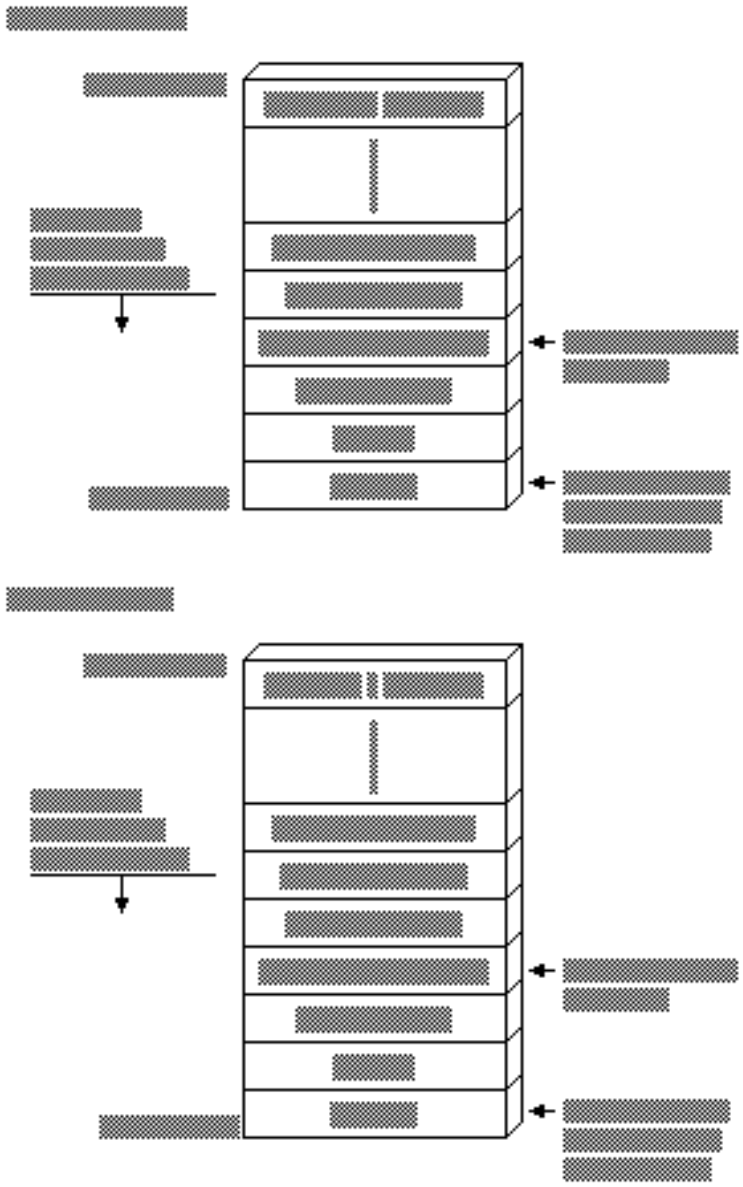

Figure 12.2 C Stack Frame

The MASM module that contains the `Power2` routine looks like this:

```
        .MODEL  small, c

Power2  PROTO C factor:SWORD, power:SWORD
        .CODE

Power2  PROC  C factor:SWORD, power:SWORD
        mov     ax, factor    ; Load Arg1 into AX
        mov     cx, power     ; Load Arg2 into CX
        shl     ax, cl        ; AX = AX * (2 to power of CX)
                              ; Leave return value in AX
        ret
Power2  ENDP
        END
```

The MASM procedure declaration for the `Power2` routine specifies the C *langtype* and the parameters expected by the procedure. The *langtype* specifies the calling and naming conventions for the interface between MASM and C. The routine is public by default. When the C module calls `Power2`, it passes two arguments, `3` and `5` by value.

Using the C Startup Code

This section explains how to write an assembly-language program that can call C library functions. It links with the C startup module, which performs the necessary initialization required by the library functions.

You must follow these steps when writing such a program:

1. Specify the C convention in the **.MODEL** statement.
2. Include the following (optional) statement to note linkage with the C startup module:

   ```
   EXTERN  _acrtused:abs
   ```

3. Prototype or declare as external all C functions the program references.
4. Include a public procedure called `main` in your assembly-language module. The C startup code calls `_main` (which is why all C programs begin with a `main` function). This procedure serves as the effective entry point for your program.
5. Omit an entry point in the program's **END** directive. The C startup code serves as the true entry point when the program runs.
6. Assemble with ML's /Cx switch to preserve the case of nonlocal names.

The following example serves as a template for these steps. The program calls the C run-time function **printf** to display two variables.

```
.MODEL  small, c                        ; Step 1: declare C conventions
EXTERN  _acrtused:abs                   ; Step 2: bring in C startup
        .
        .
        .
printf  PROTO   NEAR,                   ; Step 3: prototype
                pstring:NEAR PTR BYTE,  ;         external C
                num1:WORD, num2:VARARG  ;         routines
        .DATA
format  BYTE    '%i    %i', 13, 0

        .CODE

main    PROC    PUBLIC                  ; Step 4: C startup calls here
        .
        .
        .
        INVOKE  printf, OFFSET format, ax, bx
        .
        .
        .
        END                             ; Step 5: no label on END
```

The C++/MASM Interface

C++ can apply a protocol called a "linkage specification" to mixed-language procedures. This lets you link C++ code in the same way as C code. All information in the preceding section applies when linking assembly-language and C++ routines through the C linkage specification.

The C linkage specification forces the C++ compiler to adopt C conventions—which are not the same as C++ conventions—for listed routines. Since MASM does not specifically support C++ conventions, set the C linkage specification in your C++ code for all mixed-language routines, as shown here:

extern "C" *declaration*

where *declaration* is the prototype of an exported C++ function or an imported assembly-language procedure. You can bracket a list of declarations:

```
extern "C"
{
    int    WriteLine( short attr, char *string );
    void   GoExit( int err );
}
```

or apply the specification to individual prototypes:

```
extern "C" int    WriteLine( short attr, char *string );
extern "C" void   GoExit( int err );
```

Note the syntax remains the same whether `WriteLine` and `GoExit` are exported C++ functions or imported assembly-language routines. The linkage specification applies only to called routines, not to external variables. Use the **extern** keyword (without the **"C"**) as you normally would when identifying objects external to the C++ module.

The FORTRAN/MASM Interface

This section summarizes the specific details important to calling FORTRAN procedures or receiving arguments from FORTRAN routines that call MASM routines. It includes a sample MASM and FORTRAN module.

A FORTRAN procedure follows the Pascal calling convention by default. This convention passes arguments in the order listed, and the calling procedure removes the arguments from the stack. The naming convention converts all exported names to uppercase.

Compatible Data Types

This list shows the FORTRAN data types that are equivalent to the MASM 6.1 data types.

FORTRAN Type	Equivalent MASM Type
CHARACTER*1	**BYTE**
INTEGER*1	**SBYTE**
INTEGER*2	**SWORD**
REAL*4	**REAL4**
INTEGER*4	**SDWORD**
REAL*8, **DOUBLE PRECISION**	**REAL8**

Naming Restrictions

FORTRAN allows 31 characters for identifier names. A digit or an underscore cannot be the first character in an identifier name.

Argument-Passing Defaults

By default, FORTRAN passes arguments by reference as far addresses if the FORTRAN module is compiled in large or huge memory model. It passes them as near addresses if the FORTRAN module is compiled in medium model. Versions of FORTRAN prior to Version 4.0 always require large model.

The FORTRAN compiler passes an argument by value when declared with the **VALUE** attribute. This declaration can occur either in a FORTRAN **INTERFACE** block (which determines how to pass an argument) or in a function or subroutine declaration (which determines how to receive an argument).

In FORTRAN you can apply the **NEAR** (or **FAR**) attribute to reference parameters. These keywords override the default. They have no effect when they specify the same method as the default.

Changing the Calling Convention

A call to a FORTRAN function or subroutine declared with the **PASCAL** or **C** attribute passes all arguments by value in the 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 the functions and subroutines described by **INTERFACE** blocks.

Array Storage

When you declare FORTRAN arrays, you can specify any integer for the lower bound (the default is 1). The FORTRAN compiler stores all arrays in column-major order—that is, the leftmost subscript increments most rapidly. For example, the first seven elements of an array defined as `A[3,4]` are stored as

```
A[1,1], A[2,1], A[3,1], A[1,2], A[2,2], A[3,2], A[1,3]
```

String Format

FORTRAN stores strings as a series of bytes at a fixed location in memory, with no delimiter at the end of the string. When passing a variable-length FORTRAN string to another language, you need to devise a method by which the target routine can find the end of the string.

Consider the string declared as

```
CHARACTER*14 MSG
MSG = 'String of text'
```

The string is stored in 14 bytes of memory like this:

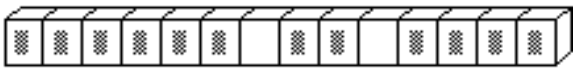

Figure 12.3 FORTRAN String Format

Strings are passed by reference. Although FORTRAN 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. However, fixed-length strings can be passed if the FORTRAN **INTERFACE** statement declares the length of the string in advance.

External Data

FORTRAN routines can directly access external data. In FORTRAN you can declare data to be external by adding the **EXTERN** attribute to the data declaration. You can also access a FORTRAN variable from MASM if it is declared in a **COMMON** block.

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.

Structure Alignment

By default, FORTRAN uses word alignment (unpacked storage) for all data objects larger than 1 byte. This storage method specifies that occasional bytes may be added as padding, so that word and doubleword objects start on an even boundary. In addition, all nested structures and records start on a word boundary. The MASM default is byte-alignment, so you should specify an *alignment* of 2 for MASM structures or use the /Zp1 option when compiling in FORTRAN.

Compiling and Linking

Use the same memory model for the MASM and FORTRAN modules.

Returning Values

You must use a special convention to return floating-point values, records, user-defined types, arrays, and values larger than 4 bytes to a FORTRAN module from an assembly procedure. The FORTRAN module creates space in the stack segment to hold the actual return value. When the call to the assembly procedure is made, an extra parameter is passed. This parameter is the last one pushed. The segment address of the return value is contained in SS.

In the assembly procedure, put the data for the return value at the location pointed to by the return value offset. Then 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.

Structures, Records, and User-Defined Data Types

The FORTRAN structure variable, defined with the **STRUCTURE** keyword and declared with the **RECORD** statement, is equivalent to the Pascal **RECORD** and the C **struct**. You can pass structures as arguments by value or by reference (the default).

The FORTRAN types **COMPLEX*8** and **COMPLEX*16** are not directly implemented in MASM. However, you can write structures that are equivalent. The type **COMPLEX*8** has two fields, both of which are 4-byte floating-point numbers; the first contains the real component, and the second contains the imaginary component. The type **COMPLEX** is equivalent to the type **COMPLEX*8**.

The type **COMPLEX*16** is similar to **COMPLEX*8**. The only difference is that each field of the former contains an 8-byte floating-point number.

A FORTRAN **LOGICAL*2** is stored as a 1-byte indicator value (1=true, 0=false) followed by an unused byte. A FORTRAN **LOGICAL*4** is stored as a 1-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 MASM structure with the appropriate fields.

Varying Number of Arguments

In FORTRAN, you can call routines with a variable number of arguments 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 arguments is possible only with the **C** calling convention. The **VARYING** attribute prevents FORTRAN from enforcing a matching number of parameters.

Pointers and Addresses

FORTRAN programs can determine near and far addresses with the **LOCNEAR** and **LOCFAR** functions. Store the result as **INTEGER*2** (with the **LOCNEAR** function) or as **INTEGER*4** (with the **LOCFAR** function). If you pass the result of **LOCNEAR** or **LOCFAR** to another language, be sure to pass by value.

Example

In the following example, the FORTRAN module calls an assembly procedure that calculates `A*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.

```
INTERFACE TO INTEGER*2 FUNCTION POWER2(A, B)
INTEGER*2 A, B
END

PROGRAM MAIN
INTEGER*2 POWER2
INTEGER*2 A, B
A = 3
B = 5
WRITE (*, *) '3 TIMES 2 TO THE B OR 5 IS ',POWER2(A, B)
END
```

To understand the assembly procedure, consider how the parameters are placed on the stack, as illustrated in Figure 12.4.

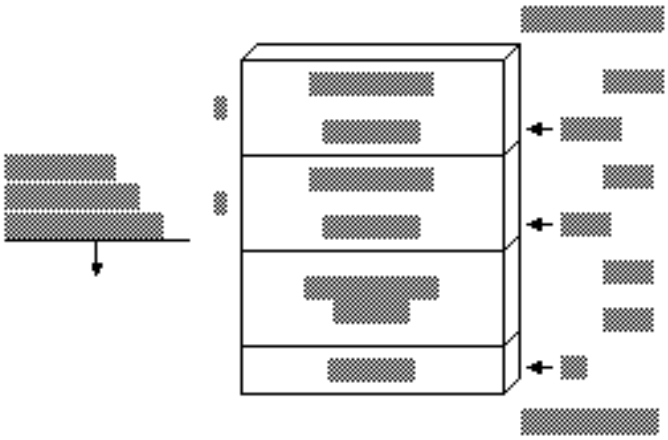

Figure 12.4 FORTRAN Stack Frame

Figure 12.4 assumes that the FORTRAN module is compiled in large model. If you compile the FORTRAN module in medium model, then each argument is passed as a 2-byte, not 4-byte, address. The return address is 4 bytes long because procedures called from FORTRAN must always be **FAR**.

The assembler code looks like this:

```
        .MODEL LARGE, FORTRAN

Power2  PROTO   FORTRAN, pFactor:FAR PTR SWORD, pPower:FAR PTR SWORD

        .CODE

Power2  PROC    FORTRAN, pFactor:FAR PTR SWORD, pPower:FAR PTR SWORD

        les     bx, pFactor     ; ES:BX points to factor
        mov     ax, es:[bx]     ; AX = value of factor
        les     bx, pPower      ; ES:BX points to power
        mov     cx, es:[bx]     ; CX = value of power
        shl     ax, cl          ; Multiply by 2^power
        ret                     ; Return result in AX
Power2  ENDP
        END
```

The Basic/MASM Interface

This section explains how to call MASM procedures or functions from Basic and how to receive Basic arguments for the MASM procedure. Pascal is the default naming and calling convention, so all lowercase letters are converted to uppercase. Routines defined with the **FUNCTION** keyword return values, but routines defined with **SUB** do not. Basic **DEF FN** functions and **GOSUB** routines cannot be called from another language.

The information provided pertains to Microsoft's Basic and QuickBasic compilers. Differences between the two compilers are noted when necessary.

Compatible Data Types

The following list shows the Basic data types that are equivalent to the MASM 6.1 data types.

Basic Type	Equivalent MASM Type
STRING*1	**WORD**
INTEGER (X%)	**SWORD**
SINGLE (X!)	**REAL4**
LONG (X&), CURRENCY	**SDWORD**
DOUBLE (X#)	**REAL8**

Naming Conventions

Basic recognizes up to 40 characters of a name. In the object code, Basic also drops any of its reserved characters: **%**, **&**, **!**, **#**, **@**, **&**.

Argument-Passing Defaults

Basic can pass data in several ways and can receive it by value or by near reference. By default, Basic arguments are passed by near reference as 2-byte addresses. To pass a near address, pass only the offset; if you need to pass a far address, pass the segment and offset separately as integer arguments. Pass the segment address first, unless you have specified C compatibility with the **CDECL** keyword.

Basic passes each argument in a call by far reference when **CALLS** is used to invoke a routine. You can also use **SEG** to modify a parameter in a preceding **DECLARE** statement so that Basic passes that argument by far reference. To pass any other variable type by value, apply the **BYVAL** keyword to the argument in the **DECLARE** statement. You cannot pass arrays and user-defined types by value.

```
DECLARE SUB Test(BYVAL a%, b%, SEG c%)

CALL Test(x%, y%, z%)
CALLS Test(x%, y%, z%)
```

This **CALL** statement passes the first argument (`a%`) by value, the second argument (`b%`) by near reference, and the third argument (`c%`) by far reference. The statement

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

passes each argument by far reference.

Changing the Calling Convention

Including the **CDECL** keyword in the Basic **DECLARE** statement enables the C calling and naming conventions. This also allows a call to a MASM procedure with a varying number of arguments.

Array Storage

The **DIM** statement sets the number of dimensions for a Basic array and also sets the array's maximum subscript value. In the array declaration `DIM x(a,b)`, the upper bounds (the maximum number of values possible) of the array are `a` and `b`. The default lower bound is 0. The default upper bound for an array subscript is 10.

The default for column storage in Basic is column-major order, as in FORTRAN. For an array defined as `DIM Arr%(3,3)`, reference the last element as `Arr%(3,3)`. The first five elements of `Arr (3,3)` are

```
Arr(0,0), Arr(1,0), Arr(2,0), Arr(0,1), Arr(1,1)
```

When you pass an array from Basic to a language that stores arrays in row-major order, use the command-line option /R when compiling the Basic module.

Most Microsoft languages permit you to reference arrays directly. Basic uses an array descriptor, however, which is similar in some respects to a Basic string descriptor. The array descriptor is necessary because Basic handles memory allocation for arrays dynamically, and thus may shift the location of the array in memory.

A reference to an array in Basic is really a near reference to an array descriptor. Array descriptors are always in DGROUP, even though the data may be in far memory. Array descriptors contain information about type, dimensions, and memory locations of data. You can safely pass arrays to MASM routines only if you follow three rules:

- Pass the array's address by applying the **VARPTR** function to the first element of the Basic 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 MASM routine that receives the array should not call back to one of the calling program's routines before it has finished processing the array. Changing data within the caller's heap—even data unrelated to the array—may change the array's location in the heap. This would invalidate any further work the called routine performs, since the routine would be operating on the array's old location.
- Basic can pass any member of an array by value. When passing individual array elements, these restrictions do not apply.

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.

String Format

Basic maintains a 4-byte string descriptor for each string, as shown in the following. The first field of the string descriptor contains a 2-byte integer indicating the length of the actual string text. The second field contains the offset address of this text within the caller's data segment.

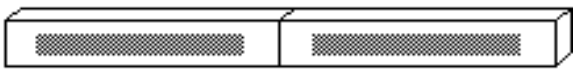

Figure 12.5 Basic String Descriptor Format

An assembly-language procedure can store a Basic string descriptor as a simple structure, like this:

```
DESC    STRUCT
  len   WORD    ?       ; Length of string
  off   WORD    ?       ; Offset of string
DESC    ENDS

string  BYTE    "This text referenced by a string descriptor"
sdesc   DESC    (LENGTHOF string, string)
```

Version 7.0 or later of the Microsoft Basic Compiler provides new functions that access string descriptors. These functions simplify the process of sharing Basic string data with routines written in other languages.

Earlier versions of Basic offer the **LEN** (Length) and **SADD** (String Address) functions, which together obtain the information stored in a string descriptor. **LEN** returns the length of a string in bytes. **SADD** returns the offset address of a string in the data segment. The caller must provide both pieces of information so the called procedure can locate and read the entire string. The address returned by **SADD** is declared as type **INTEGER** but is actually equivalent to a C near pointer.

If you need to pass the far address of a string, use the **SSEGADD** (String Segment Address) function of Microsoft Basic version 7.0 or later. You can also determine the segment address of the first element with **VARSEG**.

External Data

Declaring global data in Basic follows the same two-step process as in other languages:

1. Declare shareable data in Basic with the **COMMON** statement.
2. Identify the shared variables in your assembly-language procedures with the **EXTERN** keyword. Place the **EXTERN** statement outside of a code or data segment when declaring far data.

Structure Alignment

Basic packs user-defined types. For MASM structures to be compatible, select byte-alignment.

Compiling and Linking

Always use medium model in assembly-language procedures linked with Basic modules. If you are listing other libraries on the LINK command line, specify Basic libraries first. (There are differences between the QBX and command-line compilation. See your Basic documentation.)

Returning Values

Basic follows the usual convention of returning values in AX or DX:AX. If the value is not floating point, an array, or a structured type, or if it is less than 4 bytes long, then the 2-byte integers should be returned from the MASM procedure in AX and 4-byte integers should be returned in DX:AX. For all other types, return the near offset in AX.

User-Defined Data Types

The Basic **TYPE** statement defines structures composed of individual fields. These types are equivalent to the C **struct**, FORTRAN record (declared with the **STRUCTURE** keyword), and Pascal **Record** types.

You can use any of the Basic data types except variable-length strings or dynamic arrays in a user-defined type. Once defined, Basic types can be passed only by reference.

Varying Number of Arguments

You can vary the number of arguments in Basic when you change the calling convention with **CDECL**. To call a function with a varying number of arguments, you also need to suppress the type checking that normally forces a call to be made with a fixed number of arguments. In Basic, you can remove this type checking by omitting a parameter list from the **DECLARE** statement.

Pointers and Addresses

VARSEG returns a variable's segment address, and **VARPTR** returns a variable's offset address. These intrinsic Basic functions enable your program to pass near or far addresses.

Example

This example calls the `Power2` procedure in the MASM 6.1 module.

```
DEFINT A-Z

DECLARE FUNCTION Power2 (A AS INTEGER, B AS INTEGER)
PRINT "3 times 2 to the power of 5 is ";
PRINT Power2(3, 5)

END
```

The first argument, `A`, is higher in memory than `B` because Basic pushes arguments in the same order in which they appear.

Figure 12.6 shows how the arguments are placed on the stack.

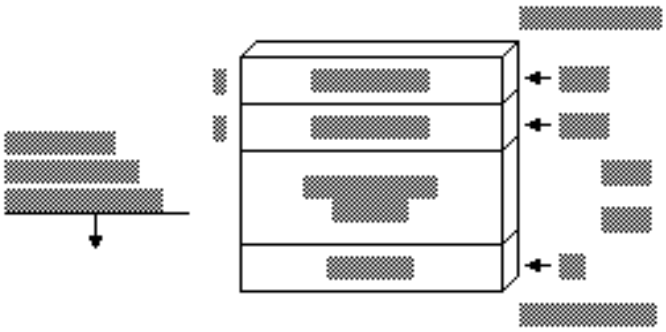

Figure 12.6 Basic Stack Frame

The assembly procedure can be written as follows:

```
        .MODEL  medium

Power2  PROTO   PASCAL, factor:PTR WORD, power:PTR WORD
        .CODE
Power2  PROC    PASCAL, factor:PTR WORD, power:PTR WORD

        mov     bx, WORD PTR factor     ; BX points to factor
        mov     ax, [bx]                ; Load factor into AX
        mov     bx, WORD PTR power      ; BX points to power
        mov     cx, [bx]                ; Load power into CX
        shl     ax, cl                  ; AX = AX * (2 to power of CX)
        ret
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. The return address is 4 bytes long because procedures called from Basic must be **FAR**.