# Mixed Language Programming Chapter Twelve

## 12.1 Chapter Overview

Most assembly language code doesn't appear in a stand-alone assembly language program. Instead, most assembly code is actually part of a library package that programs written in a high level language wind up calling. Although HLA makes it really easy to write standalone assembly applications, at one point or another you'll probably want to call an HLA procedure from some code written in another language or you may want to call code written in another language from HLA. This chapter discusses the mechanisms for doing this in three languages: low-level assembly (i.e., MASM or Gas), C/C++, and Delphi/Kylix. The mechanisms for other languages are usually similar to one of these three, so the material in this chapter will still apply even if you're using some other high level language.

## 12.2 Mixing HLA and MASM/Gas Code in the Same Program

It may seem kind of weird to mix MASM or Gas and HLA code in the same program. After all, they're both assembly languages and almost anything you can do with MASM or Gas can be done in HLA. So why bother trying to mix the two in the same program? Well, there are three reasons:

- You've already got a lot of code written in MASM or Gas and you don't want to convert it to HLA's syntax.
- There are a few things MASM and Gas do that HLA cannot, and you happen to need to do one of those things.
- Someone else has written some MASM or Gas code and they want to be able to call code you've written using HLA.

In this section, we'll discuss two ways to merge MASM/Gas and HLA code in the same program: via in-line assembly code and through linking object files.

### 12.2.1 In-Line (MASM/Gas) Assembly Code in Your HLA Programs

As you're probably aware, the HLA compiler doesn't actually produce machine code directly from your HLA source files. Instead, it first compiles the code to a MASM or Gas-compatible assembly language source file and then it calls MASM or Gas to assemble this code to object code. If you're interested in seeing the MASM or Gas output HLA produces, just edit the *filename*.ASM file that HLA creates after compiling your *filename*.HLA source file. The output assembly file isn't amazingly readable, but it is fairly easy to correlate the assembly output with the HLA source file.

HLA provides two mechanisms that let you inject raw MASM or Gas code directly into the output file it produces: the #ASM..#ENDASM sequence and the #EMIT statement. The #ASM..#ENDASM sequence copies all text between these two clauses directly to the assembly output file, e.g.,

```
#asm

    mov eax, 0       ;MASM/Gas syntax for MOV( 0, EAX );
    add eax, ebx     ; "     "     "  ADD( ebx, eax );

#endasm
```

The #ASM..#ENDASM sequence is how you inject in-line (MASM or Gas) assembly code into your HLA programs. For the most port there is very little need to use this feature, but in a few instances it is valuable.

  

Note, when using Gas, that HLA specifies the ".intel_syntax" diretive, so you should use Intel syntax when supplying Gas code between #asm and #endasm.

For example, if you're writing structured exception handling code under Windows, you'll need to access the double word at address FS:[0] (offset zero in the segment pointed at by the 80x86's FS segment register). Unfortunately, HLA does not support segmentation and the use of segment registers. However, you can drop into MASM for a statement or two in order to access this value:

```
#asm
    mov ebx, fs:[0]     ; Loads process pointer into EBX
#endasm
```

At the end of this instruction sequence, EBX will contain the pointer to the process information structure that Windows maintains.

HLA blindly copies all text between the #ASM and #ENDASM clauses directly to the assembly output file. HLA does not check the syntax of this code or otherwise verify its correctness. If you introduce an error within this section of your program, the assembler will report the error when HLA assembles your code by calling MASM or Gas.

The #EMIT statement also writes text directly to the assembly output file. However, this statement does not simply copy the text from your source file to the output file; instead, this statement copies the value of a string (constant) expression to the output file. The syntax for this statement is as follows:

```
#emit( string_expression );
```

This statement evaluates the expression and verifies that it's a string expression. Then it copies the string data to the output file. Like the #ASM/#ENDASM statement, the #EMIT statement does not check the syntax of the MASM statement it writes to the assembly file. If there is a syntax error, MASM or Gas will catch it later on when HLA assembles the output file.

When HLA compiles your programs into assembly language, it does not use the same symbols in the assembly language output file that you use in the HLA source files. There are several technical reasons for this, but the bottom line is this: you cannot easily reference your HLA identifiers in your in-line assembly code. The only exception to this rule are external identifiers. HLA external identifiers use the same name in the assembly file as in the HLA source file. Therefore, you can refer to external objects within your in-line assembly sequences or in the strings you output via #EMIT.

One advantage of the #EMIT statement is that it lets you construct MASM or Gas statements under (compile-time) program control. You can write an HLA compile-time program that generates a sequence of strings and emits them to the assembly file via the #EMIT statement. The compile-time program has access to the HLA symbol table; this means that you can extract the identifiers that HLA emits to the assembly file and use these directly, even if they aren't external objects.

The @StaticName compile-time function returns the name that HLA uses to refer to most static objects in your program. The following program demonstrates a simple use of this compile-time function to obtain the assembly name of an HLA procedure:

---

```
program emitDemo;
#include( "stdlib.hhf" )

    procedure myProc;
    begin myProc;

        stdout.put( "Inside MyProc" nl );

    end myProc;

begin emitDemo;

    ?stmt:string := "call " + @StaticName( myProc );
```

  

```
    #emit( stmt );

end emitDemo;
```

Program 12.1 Using the @StaticName Function

This example creates a string value (*stmt*) that contains something like "call ?741_myProc" and emits this assembly instruction directly to the source file ("?741_myProc" is typical of the type of name mangling that HLA does to static names it writes to the output file). If you compile and run this program, it should display "Inside MyProc" and then quit. If you look at the assembly file that HLA emits, you will see that it has given the *myProc* procedure the same name it appends to the CALL instruction[1].

The @StaticName function is only valid for static symbols. This includes STATIC, READONLY, and STORAGE variables, procedures, and iterators. It does not include VAR objects, constants, macros, class iterators, or methods.

You can access VAR variables by using the [EBP+offset] addressing mode, specifying the offset of the desired local variable. You can use the @offset compile-time function to obtain the offset of a VAR object or a parameter. The following program demonstrates how to do this:

```
program offsetDemo;
#include( "stdlib.hhf" )

var
    i:int32;

begin offsetDemo;

    mov( -255, i );
    ?stmt := "mov eax, [ebp+(" + string( @offset( i )) + ")]";
    #print( "Emitting '", stmt, "'" )
    #emit( stmt );
    stdout.put( "eax = ", (type int32 eax), nl );

end offsetDemo;
```

Program 12.2 Using the @Offset Compile-Time Function

This example emits the statement "mov eax, [ebp+(-8)]" to the assembly language source file. It turns out that -8 is the offset of the *i* variable in the offsetDemo program's activation record.

Of course, the examples of #EMIT up to this point have been somewhat ridiculous since you can achieve the same results by using HLA statements. One very useful purpose for the #emit statement, however, is to create some instructions that HLA does not support. For example, as of this writing HLA does not support the LES instruction because you can't really use it under most 32-bit operating systems. However, if

1. HLA may assign a different name that "?741_myProc" when you compile the program. The exact symbol HLA chooses varies from version to version of the assembler (it depends on the number of symbols defined prior to the definition of *myProc*. In this example, there were 741 static symbols defined in the HLA Standard Library before the definition of *myProc*.

  

you found a need for this instruction, you could easily write a macro to emit this instruction and appropriate operands to the assembly source file. Using the #EMIT statement gives you the ability to reference HLA objects, something you cannot do with the #ASM..#ENDASM sequence.

## 12.2.2 Linking MASM/Gas-Assembled Modules with HLA Modules

Although you can do some interesting things with HLA's in-line assembly statements, you'll probably never use them. Further, future versions of HLA may not even support these statements, so you should avoid them as much as possible even if you see a need for them. Of course, HLA does most of the stuff you'd want to do with the #ASM/#ENDASM and #EMIT statements anyway, so there is very little reason to use them at all. If you're going to combine MASM/Gas (or other assembler) code and HLA code together in a program, most of the time this will occur because you've got a module or library routine written in some other assembly language and you would like to take advantage of that code in your HLA programs. Rather than convert the other assembler's code to HLA, the easy solution is to simply assemble that other code to an object file and link it with your HLA programs.

Once you've compiled or assembled a source file to an object file, the routines in that module are callable from almost any machine code that can handle the routines' calling sequences. If you have an object file that contains a SQRT function, for example, it doesn't matter whether you compiled that function with HLA, MASM, TASM, NASM, Gas, or even a high level language; if it's object code and it exports the proper symbols, you can call it from your HLA program.

Compiling a module in MASM or Gas and linking that with your HLA program is little different than linking other HLA modules with your main HLA program. In the assembly source file you will have to export some symbols (using the PUBLIC directive in MASM or the .GLOBAL directive in Gas) and in your HLA program you've got to tell HLA that those symbols appear in a separate module (using the EXTERNAL option).

Since the two modules are written in assembly language, there is very little language imposed structure on the calling sequence and parameter passing mechanisms. If you're calling a function written in MASM or Gas from your HLA program, then all you've got to do is to make sure that your HLA program passes parameters in the same locations where the MASM/Gas function is expecting them.

About the only issue you've got to deal with is the case of identifiers in the two programs. By default, MASM and Gas are case insensitive. HLA, on the other hand, enforces case neutrality (which, essentially, means that it is case sensitive). If you're using MASM, there is a MASM command line option ("/Cp") that tells MASM to preserve case in all public symbols. It's a real good idea to use this option when assembling modules you're going to link with HLA so that MASM doesn't mess with the case of your identifiers during assembly.

Of course, since MASM and Gas process symbols in a case sensitive manner, it's possible to create two separate identifiers that are the same except for alphabetic case. HLA enforces case neutrality so it won't let you (directly) create two different identifiers that differ only in case. In general, this is such a bad programming practice that one would hope you never encounter it (and God forbid you actually do this yourself). However, if you inherit some MASM or Gas code written by a C hacker, it's quite possible the code uses this technique. The way around this problem is to use two separate identifiers in your HLA program and use the extended form of the EXTERNAL directive to provide the external names. For example, suppose that in MASM you have the following declarations:

```
            public  AVariable
            public  avariable
                .
                .
                .
            .data
AVariable   dword    ?
avariable   byte     ?
```

  

If you assemble this code with the "/Cp" or "/Cx" (total case sensitivity) command line options, MASM will emit these two external symbols for use by other modules. Of course, were you to attempt to define variables by these two names in an HLA program, HLA would complain about a duplicate symbol definition. However, you can connect two different HLA variables to these two identifiers using code like the following:

```
static
    AVariable: dword; external( "AVariable" );
    AnotherVar: byte; external( "avariable" );
```

HLA does not check the strings you supply as parameters to the EXTERNAL clause. Therefore, you can supply two names that are the same except for case and HLA will not complain. Note that when HLA calls MASM to assemble it's output file, HLA specifies the "/Cp" option that tells MASM to preserve case in public and global symbols. Of course, you would use this same technique in Gas if the Gas programmer has exported two symbols that are identical except for case.

The following program demonstrates how to call a MASM subroutine from an HLA main program:

```
// To compile this module and the attendant MASM file, use the following
// command line:
//
//      ml -c masmupper.masm
//      hla masmdemo1.hla masmupper.obj
//
//  Sorry about no make file for this code, but these two files are in
//  the HLA Vol4/Ch12 subdirectory that has it's own makefile for building
//  all the source files in the directory and I wanted to avoid confusion.

program MasmDemo1;
#include( "stdlib.hhf" )

    // The following external declaration defines a function that
    // is written in MASM to convert the character in AL from
    // lower case to upper case.

    procedure masmUpperCase( c:char in al ); external( "masmUpperCase" );

static
    s: string := "Hello World!";

begin MasmDemo1;

    stdout.put( "String converted to uppercase: '" );
    mov( s, edi );
    while( mov( [edi], al ) <> #0 ) do

        masmUpperCase( al );
        stdout.putc( al );
        inc( edi );

    endwhile;
    stdout.put( "'" nl );


end MasmDemo1;
```

  

Program 12.3 Main HLA Program to Link with a MASM Program

```
; MASM source file to accompany the MasmDemo1.HLA source
; file.  This code compiles to an object module that
; gets linked with an HLA main program.  The function
; below converts the character in AL to upper case if it
; is a lower case character.

        .586
        .model  flat, pascal

        .code
        public  masmUpperCase
masmUpperCase   proc    near32
        .if al >= 'a' && al <= 'z'
        and al, 5fh
        .endif
        ret
masmUpperCase   endp
        end
```

Program 12.4 Calling a MASM Procedure from an HLA Program: MASM Module

It is also possible to call an HLA procedure from a MASM or Gas program (this should be obvious since HLA compiles its source code to an assembly source file and that assembly source file can call HLA procedures such as those found in the HLA Standard Library). There are a few restrictions when calling HLA code from some other language. First of all, you can't easily use HLA's exception handling facilities in the modules you call from other languages (including MASM or Gas). The HLA main program initializes the exception handling system; this initialization is probably not done by your non-HLA assembly programs. Further, the HLA main program exports a couple of important symbols needed by the exception handling subsystem; again, it's unlikely your non-HLA main assembly program provides these public symbols. In the volume on Advanced Procedures this text will discuss how to deal with HLA's Exception Handling subsystem. However, that topic is a little too advanced for this chapter. Until you get to the point you can write code in MASM or Gas to properly set up the HLA exception handling system, you should not execute any code that uses the TRY..ENDTRY, RAISE, or any other exception handling statements.

> **Warning**; a large percentage of the HLA Standard Library routines include exception handling statements or call other routines that use exception handling statements. Unless you've set up the HLA exception handling subsystem properly, you should not call any HLA Standard Library routines from non-HLA programs.

Other than the issue of exception handling, calling HLA procedures from standard assembly code is really easy. All you've got to do is put an EXTERNAL prototype in the HLA code to make the symbol you wish to access public and then include an EXTERN (or EXTERNDEF) statement in the MASM/Gas source file to provide the linkage. Then just compile the two source files and link them together.

About the only issue you need concern yourself with when calling HLA procedures from assembly is the parameter passing mechanism. Of course, if you pass all your parameters in registers (the best place), then communication between the two languages is trivial. Just load the registers with the appropriate param-

  

eters in your MASM/Gas code and call the HLA procedure. Inside the HLA procedure, the parameter values will be sitting in the appropriate registers (sort of the converse of what happened in Program 12.4).

If you decide to pass parameters on the stack, note that HLA normally uses the PASCAL language calling model. Therefore, you push parameters on the stack in the order they appear in a parameter list (from left to right) and it is the called procedure's responsibility to remove the parameters from the stack. Note that you can specify the PASCAL calling convention for use with MASM's INVOKE statement using the ".model" directive, e.g.,

```
.586
.model  flat, pascal
   .
   .
   .
```

Of course, if you manually push the parameters on the stack yourself, then the specific language model doesn't really matter. Gas users, of course, don't have the INVOKE statement, so they have to manually push the parameters themselves anyway.

This section is not going to attempt to go into gory details about MASM or Gas syntax. There is an appendix in this text that contrasts the HLA language with MASM (and Gas when using the ".intel_syntax" directive); you should be able to get a rough idea of MASM/Gas syntax from that appendix if you're completely unfamiliar with these assemblers. Another alternative is to read a copy of the DOS/16-bit edition of this text that uses the MASM assembler. That text describes MASM syntax in much greater detail, albeit from a 16-bit perspective. Finally, this section isn't going to go into any further detail because, quite frankly, the need to call MASM or Gas code from HLA (or vice versa) just isn't that great. After all, most of the stuff you can do with MASM and Gas can be done directly in HLA so there really is little need to spend much more time on this subject. Better to move on to more important questions, like how do you call HLA routines from C or Pascal...

## 12.3 Programming in Delphi/Kylix and HLA

Delphi is a marvelous language for writing Win32 GUI-based applications. Kylix is the companion product that runs under Linux. Their support for Rapid Application Design (RAD) and visual programming is superior to almost every other Windows or Linux programming approach available. However, being Pascal-based, there are some things that just cannot be done in Delphi/Kylix and many things that cannot be done as efficiently in Delphi/Kylix as in assembly language. Fortunately, Delphi/Kylix lets you call assembly language procedures and functions so you can overcome Delphi's limitations.

Delphi provides two ways to use assembly language in the Pascal code: via a built-in assembler (BASM) or by linking in separately compiled assembly language modules. The built-in "Borland Assembler" (BASM) is a very weak Intel-syntax assembler. It is suitable for injecting a few instructions into your Pascal source code or perhaps writing a very short assembly language function or procedure. It is not suitable for serious assembly language programming. If you know Intel syntax and you only need to execute a few machine instructions, then BASM is perfect. However, since this is a text on assembly language programming, the assumption here is that you want to write some serious assembly code to link with your Pascal/Delphi code. To do that, you will need to write the assembly code and compile it with a different assembler (e.g., HLA) and link the code into your Delphi application. That is the approach this section will concentrate on. For more information about BASM, check out the Delphi documentation.

Before we get started discussing how to write HLA modules for your Delphi programs, you must understand two very important facts:

> HLA's exception handling facilities are not directly compatible with Delphi's. This means that you cannot use the TRY..ENDTRY and RAISE statements in the HLA code you intend to link to a Delphi program. This also means that you cannot call library functions

  

that contain such statements. Since the HLA Standard Library modules use exception handling statements all over the place, this effectively prevents you from calling HLA Standard Library routines from the code you intend to link with Delphi[2].

Although you can write console applications with Delphi, 99% of Delphi applications are GUI applications. You cannot call console-related functions (e.g., stdin.xxxx or stdout.xxxx) from a GUI application. Even if HLA's console and standard input/output routines didn't use exception handling, you wouldn't be able to call them from a standard Delphi application.

Given the rich set of language features that Delphi supports, it should come as no surprise that the interface between Delphi's Object Pascal language and assembly language is somewhat complex. Fortunately there are two facts that reduce this problem. First, HLA uses many of the same calling conventions as Pascal; so much of the complexity is hidden from sight by HLA. Second, the other complex stuff you won't use very often, so you may not have to bother with it.

**Note**: the following sections assume you are already familiar with Delphi programming. They make no attempt to explain Delphi syntax or features other than as needed to explain the Delphi assembly language interface. If you're not familiar with Delphi, you will probably want to skip this section.

### 12.3.1 Linking HLA Modules With Delphi Programs

The basic unit of interface between a Delphi program and assembly code is the procedure or function. That is, to combine code between the two languages you will write procedures in HLA (that correspond to procedures or functions in Delphi) and call these procedures from the Delphi program. Of course, there are a few mechanical details you've got to worry about, this section will cover those.

To begin with, when writing HLA code to link with a Delphi program you've got to place your HLA code in an HLA UNIT. An HLA PROGRAM module contains start up code and other information that the operating system uses to determine where to begin program execution when it loads an executable file from disk. However, the Delphi program also supplies this information and specifying two starting addresses confuses the linker, therefore, you must place all your HLA code in a UNIT rather than a PROGRAM module.

Within the HLA UNIT you must create EXTERNAL procedure prototypes for each procedure you wish to call from Delphi. If you prefer, you can put these prototype declarations in a header file and #INCLUDE them in the HLA code, but since you'll probably only reference these declarations from this single file, it's okay to put the EXTERNAL prototype declarations directly in the HLA UNIT module. These EXTERNAL prototype declarations tell HLA that the associated functions will be public so that Delphi can access their names during the link process. Here's a typical example:

```
unit LinkWithDelphi;

    procedure prototype; external;

    procedure prototype;
    begin prototype;

        << Code to implement prototype's functionality >>

    end prototype;

end LinkWithDelphi;
```

After creating the module above, you'd compile it using HLA's "-s" (compile to assembly only) command line option. This will produce an ASM file. Were this just about any other language, you'd then assemble

2. Note that the HLA Standard Library source code is available; feel free to modify the routines you want to use and remove any exception handling statements contained therein.

  

the ASM file with MASM. Unfortunately, Delphi doesn't like OBJ files that MASM produces. For all but the most trivial of assembly modules, Delphi will reject the MASM's output. Borland Delphi expects external assembly modules to be written with Borland's assembler, TASM32.EXE (the 32-bit Turbo Assembler). Fortunately, as of HLA v1.26, HLA provides an option to produce TASM output that is compatible with TASM v5.3 and later. Unfortunately, Borland doesn't really sell TASM anymore; the only way to get a copy of TASM v5.3 is to obtain a copy of Borlands C++ Builder Professional system which includes TASM32 v5.3. If you don't own Borland C++ and really have no interest in using C++ Builder, Borland has produced an evaluation disk for C++ Builder that includes TASM 5.3. Note that earlier versions of TASM32 (e.g., v5.0) do not support MMX and various Pentium-only instructions, you really need TASM v5.3 if you want ot use the MASM output.

Here are all the commands to compile and assemble the module given earlier:

```
hla -c -tasm -omf LinkWithDelphi.hla
```

Of course, if you don't like typing this long command to compile and assemble your HLA code, you can always create a make file or a batch file that will let you do both operations with a single command. See the chapter on Managing Large Programs for more details (see "Make Files" on page 578).

After creating the module above, you'd compile it using HLA's "-c" (compile to object only) command line option. This will produce an object (".o") file.

Once you've created the HLA code and compiled it to an object file, the next step is to tell Delphi that it needs to call the HLA/assembly code. There are two steps needed to achieve this: You've got to inform Delphi that a procedure (or function) is written in assembly language (rather than Pascal) and you've got to tell Delphi to link in the object file you've created when compiling the Delphi code.

The second step above, telling Delphi to include the HLA object module, is the easiest task to achieve. All you've got to do is insert a compiler directive of the form "{$L *objectFileName*.obj }" in the Delphi program before declaring and calling your object module. A good place to put this is after the **implementation** reserved word in the module that calls your assembly procedure. The code examples a little later in this section will demonstrate this.

The next step is to tell Delphi that you're supplying an external procedure or function. This is done using the Delphi EXTERNAL directive on a procedure or function prototype. For example, a typical external declaration for the *prototype* procedure appearing earlier is

```
procedure prototype; external;  // This may look like HLA code, but it's
                                // really Delphi code!
```

As you can see here, Delphi's syntax for declaring external procedures is nearly identical to HLA's (in fact, in this particular example the syntax is identical). This is not an accident, much of HLA's syntax was borrowed directly from Pascal.

The next step is to call the assembly procedure from the Delphi code. This is easily accomplished using standard Pascal procedure calling syntax. The following two listings provide a complete, working, example of an HLA procedure that a Delphi program can call. This program doesn't accomplish very much other than to demonstrate how to link in an assembly procedure. The Delphi program contains a form with a single button on it. Pushing the button calls the HLA procedure, whose body is empty and therefore returns immediately to the Delphi code without any visible indication that it was ever called. Nevertheless, this code does provide all the syntactical elements necessary to create and call an assembly language routine from a Delphi program.

---

```
unit LinkWithDelphi;

    procedure CalledFromDelphi; external;

    procedure CalledFromDelphi;
    begin CalledFromDelphi;
```

  

```
    end CalledFromDelphi;

end LinkWithDelphi;
```

Program 12.5 CalledFromDelphi.HLA Module Containing the Assembly Code

```
unit DelphiEx1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TDelphiEx1Form = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  DelphiEx1Form: TDelphiEx1Form;

implementation

{$R *.DFM}
{$L CalledFromDelphi.obj }

procedure CalledFromDelphi; external;

procedure TDelphiEx1Form.Button1Click(Sender: TObject);
begin

    CalledFromDelphi();

end;

end.
```

Program 12.6 DelphiEx1– Delphi Source Code that Calls an Assembly Procedure

  

The full Delphi and HLA source code for the programs appearing in Program 12.5 and Program 12.6 accompanies the HLA software distribution in the appropriate subdirectory for this chapter in the Example code module. If you've got a copy of Delphi 5 or later, you might want to load this module and try compiling it. To compile the HLA code for this example, you would use the following commands from the command prompt:

```
hla -tasm -c -omf CalledFromDelphi.hla
```

After producing the CalledFromDelphi object module with the two commands above, you'd enter the Delphi Integrated Development Environment and tell it to compile the DelphiEx1 code (i.e., you'd load the DelphiEx1Project file into Delphi and the compile the code). This process automatically links in the HLA code and when you run the program you can call the assembly code by simply pressing the single button on the Delphi form.

## 12.3.2 Register Preservation

Delphi code expects all procedures to preserve the EBX, ESI, EDI, and EBP registers. Routines written in assembly language may freely modify the contents of EAX, ECX, and EDX without preserving their values. The HLA code will have to modify the ESP register to remove the activation record (and, possibly, some parameters). Of course, HLA procedures (unless you specify the @NOFRAME option) automatically preserve and set up EBP for you, so you don't have to worry about preserving this register's value; of course, you will not usually manipulate EBP's value since it points at your procedure's parameters and local variables.

Although you can modify EAX, ECX, and EDX to your heart's content and not have to worry about preserving their values, don't get the idea that these registers are available for your procedure's exclusive use. In particular, Delphi may pass parameters into a procedure within these registers and you may need to return function results in some of these registers. Details on the further use of these registers appears in later sections of this chapter.

Whenever Delphi calls a procedure, that procedure can assume that the direction flag is clear. On return, all procedures must ensure that the direction flag is still clear. So if you manipulate the direction flag in your assembly code (or call a routine that might set the direction flag), be sure to clear the direction flag before returning to the Delphi code.

If you use any MMX instructions within your assembly code, be sure to execute the EMMS instruction before returning. Delphi code assumes that it can manipulate the floating point stack without running into problems.

Although the Delphi documentation doesn't explicitly state this, experiments with Delphi code seem to suggest that you don't have to preserve the FPU (or MMX) registers across a procedure call other than to ensure that you're in FPU mode (versus MMX mode) upon return to Delphi.

## 12.3.3 Function Results

Delphi generally expects functions to return their results in a register. For ordinal return results, a function should return a byte value in AL, a word value in AX, or a double word value in EAX. Functions return pointer values in EAX. Functions return real values in ST0 on the FPU stack. The code example in this section demonstrates each of these parameter return locations.

For other return types (e.g., arrays, sets, records, etc.), Delphi generally passes an extra VAR parameter containing the address of the location where the function should store the return result. We will not consider such return results in this text, see the Delphi documentation for more details.

  

The following Delphi/HLA program demonstrates how to return different types of scalar (ordinal and real) parameters to a Delphi program from an assembly language function. The HLA functions return boolean (one byte) results, word results, double word results, a pointer (PChar) result, and a floating point result when you press an appropriate button on the form. See the DelphiEx2 example code in the HLA/Art of Assembly examples code for the full project. Note that the following code doesn't really do anything useful other than demonstrate how to return Function results in EAX and ST0.

---

```
unit DelphiEx2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TDelphiEx2Form = class(TForm)
    BoolBtn: TButton;
    BooleanLabel: TLabel;
    WordBtn: TButton;
    WordLabel: TLabel;
    DWordBtn: TButton;
    DWordLabel: TLabel;
    PtrBtn: TButton;
    PCharLabel: TLabel;
    FltBtn: TButton;
    RealLabel: TLabel;
    procedure BoolBtnClick(Sender: TObject);
    procedure WordBtnClick(Sender: TObject);
    procedure DWordBtnClick(Sender: TObject);
    procedure PtrBtnClick(Sender: TObject);
    procedure FltBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  DelphiEx2Form: TDelphiEx2Form;

implementation

{$R *.DFM}

// Here's the directive that tells Delphi to link in our
// HLA code.

{$L ReturnBoolean.obj }
{$L ReturnWord.obj }
{$L ReturnDWord.obj }
{$L ReturnPtr.obj }
{$L ReturnReal.obj }

// Here are the external function declarations:

function ReturnBoolean:boolean; external;
```

  

```
function ReturnWord:smallint; external;
function ReturnDWord:integer; external;
function ReturnPtr:pchar; external;
function ReturnReal:real; external;


// Demonstration of calling an assembly language
// procedure that returns a byte (boolean) result.

procedure TDelphiEx2Form.BoolBtnClick(Sender: TObject);
var
    b:boolean;

begin

    // Call the assembly code and return its result:

    b := ReturnBoolean;

    // Display “true” or “false” depending on the return result.

    if( b ) then

        booleanLabel.caption := ‘Boolean result = true ‘

    else

        BooleanLabel.caption := ‘Boolean result = false’;

end;


// Demonstrate calling an assembly language function that
// returns a word result.

procedure TDelphiEx2Form.WordBtnClick(Sender: TObject);
var
    si:smallint;    // Return result here.
    strVal:string;  // Used to display return result.
begin

    si := ReturnWord();     // Get result from assembly code.
    str( si, strVal );      // Convert result to a string.
    WordLabel.caption := ‘Word Result = ‘ + strVal;

end;

// Demonstration of a call to an assembly language routine
// that returns a 32-bit result in EAX:

procedure TDelphiEx2Form.DWordBtnClick(Sender: TObject);
var
    i:integer;          // Return result goes here.
    strVal:string;      // Used to display return result.
begin

    i := ReturnDWord(); // Get result from assembly code.
    str( i, strVal );   // Convert that value to a string.
    DWordLabel.caption := ‘Double Word Result = ‘ + strVal;

end;
```

  

```
// Demonstration of a routine that returns a pointer
// as the function result.  This demo is kind of lame
// because we can't initialize anything inside the
// assembly module, but it does demonstrate the mechanism
// even if this example isn't very practical.

procedure TDelphiEx2Form.PtrBtnClick(Sender: TObject);
var
    p:pchar;    // Put returned pointer here.
begin

    // Get the pointer (to a zero byte) from the assembly code.

    p := ReturnPtr();

    // Display the empty string that ReturnPtr returns.

    PCharLabel.caption := 'PChar Result = "' + p + '"';

end;


// Quick demonstration of a function that returns a
// floating point value as a function result.

procedure TDelphiEx2Form.FltBtnClick(Sender: TObject);
var
    r:real;
    strVal:string;
begin

    // Call the assembly code that returns a real result.

    r := ReturnReal();      // Always returns 1.0

    // Convert and display the result.

    str( r:13:10, strVal );
    RealLabel.caption := 'Real Result = ' + strVal;

end;

end.
```

Program 12.7 DelphiEx2: Pascal Code for Assembly Return Results Example

```
// ReturnBooleanUnit-
//
//  Provides the ReturnBoolean function for the DelphiEx2 program.
```

  

```
unit  ReturnBooleanUnit;

// Tell HLA that ReturnBoolean is a public symbol:

procedure ReturnBoolean; external;

// Demonstration of a function that returns a byte value in AL.
// This function simply returns a boolean result that alterates
// between true and false on each call.

procedure ReturnBoolean;  @nodisplay; @noalignstack; @noframe;
static b:boolean:=false;
begin ReturnBoolean;

    xor( 1, b );    // Invert boolean status
    and( 1, b );    // Force to zero (false) or one (true).
    mov( b, al );   // Function return result comes back in AL.
    ret();

end ReturnBoolean;


end ReturnBooleanUnit;
```

Program 12.8 ReturnBoolean: Demonstrates Returning a Byte Value in AL

```
// ReturnWordUnit-
//
//  Provides the ReturnWord function for the DelphiEx2 program.


unit  ReturnWordUnit;

procedure ReturnWord; external;

procedure ReturnWord;  @nodisplay; @noalignstack; @noframe;
static w:int16 := 1234;
begin ReturnWord;

    // Increment the static value by one on each
    // call and return the new result as the function
    // return value.

    inc( w );
    mov( w, ax );
    ret();

end ReturnWord;


end ReturnWordUnit;
```

  

Program 12.9 ReturnWord: Demonstrates Returning a Word Value in AX

```
// ReturnDWordUnit-
//
//  Provides the ReturnDWord function for the DelphiEx2 program.

unit  ReturnDWordUnit;

procedure ReturnDWord; external;

// Same code as ReturnWord except this one returns a 32-bit value
// in EAX rather than a 16-bit value in AX.

procedure ReturnDWord;  @nodisplay; @noalignstack; @noframe;
static
    d:int32 := -7;
begin ReturnDWord;

    inc( d );
    mov( d, eax );
    ret();

end ReturnDWord;


end ReturnDWordUnit;
```

Program 12.10 ReturnDWord: Demonstrates Returning a DWord Value in EAX

```
// ReturnPtrUnit-
//
//  Provides the ReturnPtr function for the DelphiEx2 program.

unit  ReturnPtrUnit;

procedure ReturnPtr; external;

// This function, which is lame, returns a pointer to a zero
// byte in memory (i.e., an empty pchar string).   Although
// not particularly useful, this code does demonstrate how
// to return a pointer in EAX.

procedure ReturnPtr;  @nodisplay; @noalignstack; @noframe;
static
    stringData: byte; @nostorage;
                byte “Pchar object”, 0;

begin ReturnPtr;

    lea( eax, stringData );
    ret();
```

  

```
end ReturnPtr;


end ReturnPtrUnit;
```

Program 12.11 ReturnPtr: Demonstrates Returning a 32-bit Address in EAX

```
// ReturnRealUnit-
//
//  Provides the ReturnReal function for the DelphiEx2 program.

unit  ReturnRealUnit;

procedure ReturnReal; external;
procedure ReturnReal;  @nodisplay; @noalignstack; @noframe;
static
    realData: real80 := 1.234567890;

begin ReturnReal;

    fld( realData );
    ret();

end ReturnReal;


end ReturnRealUnit;
```

Program 12.12 ReturnReal: Demonstrates Returning a Real Value in ST0

The second thing to note is the #code, #static, etc., directives at the beginning of each file to change the segment name declarations. You'll learn the reason for these segment renaming directives a little later in this chapter.

### 12.3.4 Calling Conventions

Delphi supports five different calling mechanisms for procedures and functions: **register, pascal, cdecl, stdcall**, and **safecall**. The **register** and **pascal** calling methods are very similar except that the **pascal** parameter passing scheme always passes all parameters on the stack while the **register** calling mechanism passes the first three parameters in CPU registers. We'll return to these two mechanisms shortly since they are the primary mechanisms we'll use. The **cdecl** calling convention uses the C/C++ programming language calling convention. We'll study this scheme more in the section on interfacing C/C++ with HLA. There is no need to use this scheme when calling HLA procedures from Delphi. If you must use this scheme, then see the section on the C/C++ languages for details. The **stdcall** convention is used to call Windows API functions. Again, there really is no need to use this calling convention, so we will ignore it here. See the

Delphi documentation for more details. **Safecall** is another specialized calling convention that we will not use. See, we've already reduced the complexity from five mechanisms to two! Seriously, though, when calling assembly language routines from Delphi code that you're writing, you only need to use the **pascal** and **register** conventions.

The calling convention options specify how Delphi passes parameters between procedures and functions as well as who is responsible for cleaning up the parameters when a function or procedure returns to its caller. The **pascal** calling convention passes all parameters on the stack and makes it the procedure or function's responsibility to remove those parameters from the stack. The pascal calling convention mandates that the caller push parameters in the order the compiler encounters them in the parameter list (i.e., left to right). This is exactly the calling convention that HLA uses (assuming you don't use the "IN register" parameter option). Here's an example of a Delphi external procedure declaration that uses the **pascal** calling convention:

```
procedure UsesPascal( parm1:integer; parm2:integer; parm3:integer );
```

The following program provides a quick example of a Delphi program that calls an HLA procedure (function) using the **pascal** calling convention.

---

```
unit DelphiEx3;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    callUsesPascalBtn: TButton;
    UsesPascalLabel: TLabel;
    procedure callUsesPascalBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
{$L usespascal.obj}

function UsesPascal
(
    parm1:integer;
    parm2:integer;
    parm3:integer
):integer; pascal; external;

procedure TForm1.callUsesPascalBtnClick(Sender: TObject);
var
    i:      integer;
    strVal: string;
begin
```

  

```
    i := UsesPascal( 5, 6, 7 );
    str( i, strVal );
    UsesPascalLabel.caption := 'Uses Pascal = ' + strVal;

end;

end.
```

Program 12.13 DelphiEx3 – Sample Program that Demonstrates the **pascal** Calling Convention

```
// UsesPascalUnit-
//
//  Provides the UsesPascal function for the DelphiEx3 program.

unit  UsesPascalUnit;

// Tell HLA that UsesPascal is a public symbol:

procedure UsesPascal( parm1:int32; parm2:int32; parm3:int32 ); external;

// Demonstration of a function that uses the PASCAL calling convention.
// This function simply computes parm1+parm2-parm3 and returns the
// result in EAX.  Note that this function does not have the
// "NOFRAME" option because it needs to build the activation record
// (stack frame) in order to access the parameters.  Furthermore, this
// code must clean up the parameters upon return (another chore handled
// automatically by HLA if the "NOFRAME" option is not present).

procedure UsesPascal( parm1:int32; parm2:int32; parm3:int32 );
    @nodisplay; @noalignstack;

begin UsesPascal;

    mov( parm1, eax );
    add( parm2, eax );
    sub( parm3, eax );

end UsesPascal;


end UsesPascalUnit;
```

Program 12.14 UsesPascal – HLA Function the Previous Delphi Code Will Call

To compile the HLA code, you would use the following two commands in a command window:

```
hla -st UsesPascal.hla
```

  

```
tasm32 -mx -m9 UsesPascal.asm
```

Once you produce the .o file with the above two commands, you can get into Delphi and compile the Pascal code.

The **register** calling convention also processes parameters from left to right and requires the procedure/function to clean up the parameters upon return; the difference is that procedures and functions that use the **register** calling convention will pass their first three (ordinal) parameters in the EAX, EDX, and ECX registers (in that order) rather than on the stack. You can use HLA's "IN *register*" syntax to specify that you want the first three parameters passed in this registers, e.g.,

```
procedure UsesRegisters
(
    parm1:int32 in EAX;
    parm2:int32 in EDX;
    parm3:int32 in ECX
);
```

If your procedure had four or more parameters, you would not specify registers as their locations. Instead, you'd access those parameters on the stack. Since most procedures have three or fewer parameters, the **register** calling convention will typically pass all of a procedure's parameters in a register.

Although you can use the **register** keyword just like **pascal** to force the use of the **register** calling convention, the register calling convention is the default mechanism in Delphi. Therefore, a Delphi declaration like the following will automatically use the **register** calling convention:

```
procedure UsesRegisters
(
    parm1:integer;
    parm2:integer;
    parm3:integer
); external;
```

The following program is a modification of the previous program in this section that uses the **register** calling convention rather than the **pascal** calling convention.

---

```
unit DelphiEx4;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    callUsesRegisterBtn: TButton;
    UsesRegisterLabel: TLabel;
    procedure callUsesRegisterBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

  

```
{$R *.DFM}
{$L usesregister.obj}

function UsesRegister
(
    parm1:integer;
    parm2:integer;
    parm3:integer;
    parm4:integer
):integer; external;

procedure TForm1.callUsesRegisterBtnClick(Sender: TObject);
var
    i:      integer;
    strVal: string;
begin

    i := UsesRegister( 5, 6, 7, 3 );
    str( i, strVal );
    UsesRegisterLabel.caption := 'Uses Register = ' + strVal;

end;

end.
```

Program 12.15 DelphiEx4 – Using the **register** Calling Convention

```
// UsesRegisterUnit-
//
//  Provides the UsesRegister function for the DelphiEx4 program.

unit  UsesRegisterUnit;

// Tell HLA that UsesRegister is a public symbol:

procedure UsesRegister
(
    parm1:int32 in eax;
    parm2:int32 in edx;
    parm3:int32 in ecx;
    parm4:int32
);  external;

// Demonstration of a function that uses the REGISTER calling convention.
// This function simply computes (parm1+parm2-parm3)*parm4 and returns the
// result in EAX.  Note that this function does not have the
// "NOFRAME" option because it needs to build the activation record
// (stack frame) in order to access the fourth parameter.  Furthermore, this
// code must clean up the fourth parameter upon return (another chore handled
// automatically by HLA if the "NOFRAME" option is not present).

procedure UsesRegister
(
```

  

```
    parm1:int32 in eax;
    parm2:int32 in edx;
    parm3:int32 in ecx;
    parm4:int32
);  @nodisplay; @noalignstack;

begin UsesRegister;

    mov( parm1, eax );
    add( parm2, eax );
    sub( parm3, eax );
    intmul( parm4, eax );

end UsesRegister;


end UsesRegisterUnit;
```

Program 12.16 HLA Code to support the DelphiEx4 Program

To compile the HLA code, you would use the following two commands in a command window:

```
hla -st UsesRegister.hla
tasm32 -mx -m9 UsesRegister.hla
```

Once you produce the OBJ file with the above command, you can get into Delphi and compile the Pascal code.

## 12.3.5 Pass by Value, Reference, CONST, and OUT in Delphi

A Delphi program can pass parameters to a procedure or function using one of four different mechanisms: pass by value, pass by reference, CONST parameters, and OUT parameters. The examples up to this point in this chapter have all used Delphi's (and HLA's) default pass by value mechanism. In this section we'll look at the other parameter passing mechanisms.

HLA and Delphi also share a (mostly) common syntax for pass by reference parameters. The following two lines provide an external declaration in Delphi and the corresponding external (public) declaration in HLA for a pass by reference parameter using the **pascal** calling convention:

```
procedure HasRefParm( var refparm: integer ); pascal; external; // Delphi
procedure HasRefParm( var refparm: int32 ); external;           // HLA
```

Like HLA, Delphi will pass the 32-bit address of whatever actual parameter you specify when calling the *HasRefParm* procedure. Don't forget, inside the HLA code, that you must dereference this pointer to access the actual parameter data. See the chapter on Intermediate Procedures for more details (see "Pass by Reference" on page 817).

The CONST and OUT parameter passing mechanisms are virtually identical to pass by reference. Like pass by reference these two schemes pass a 32-bit address of their actual parameter. The difference is that the called procedure is not supposed to write to CONST objects since they're, presumably, constant. Conversely, the called procedure is supposed to write to an OUT parameter (and not assume that it contains any initial value of consequence) since the whole purpose of an OUT parameter is to return data from a procedure or function. Other than the fact that the Delphi compiler will check procedures and functions (written in Delphi) for compliance with these rules, there is no difference between CONST, OUT, and reference parameters. Delphi passes all such parameters by reference to the procedure or function. Note that in HLA

  

you would declare all CONST and OUT parameters as pass by reference parameters. HLA does not enforce the readonly attribute of the CONST object nor does it check for an attempt to access an uninitialized OUT parameter; those checks are the responsibility of the assembly language programmer.

As you learned in the previous section, by default Delphi uses the **register** calling convention. If you pass one of the first three parameters by reference to a procedure or function, Delphi will pass the address of that parameter in the EAX, EDX, or ECX register. This is very convenient as you can immediately apply the register indirect addressing mode without first loading the parameter into a 32-bit register.

Like HLA, Delphi lets you pass untyped parameters by reference (or by CONST or OUT). The syntax to achieve this in Delphi is the following:

```
procedure UntypedRefParm( var parm1; const parm2; out parm3 ); external;
```

Note that you do not supply a type specification for these parameters. Delphi will compute the 32-bit address of these objects and pass them on to the *UntypedRefParm* procedure without any further type checking. In HLA, you can use the VAR keyword as the data type to specify that you want an untyped reference parameter. Here's the corresponding prototype for the *UntypedRefParm* procedure in HLA:

```
procedure UntypedRefParm( var parm1:var; var parm2:var; var parm3:var );
external;
```

As noted above, you use the VAR keyword (pass by reference) when passing CONST and OUT parameters. Inside the HLA procedure it's your responsibility to use these pointers in a manner that is reasonable given the expectations of the Delphi code.

## 12.3.6 Scalar Data Type Correspondence Between Delphi and HLA

When passing parameters between Delphi and HLA procedures and functions, it's very important that the calling code and the called code agree on the basic data types for the parameters. In this section we will draw a correspondence between the Delphi scalar data types and the HLA (v1.x) data types[3].

Assembly language supports any possible data format, so HLA's data type capabilities will always be a superset of Delphi's. Therefore, there may be some objects you can create in HLA that have no counterpart in Delphi, but the reverse is not true. Since the assembly functions and procedures you write are generally manipulating data that Delphi provides, you don't have to worry too much about not being able to process some data passed to an HLA procedure by Delphi[4].

Delphi provides a wide range of different integer data types. The following table lists the Delphi types and the HLA equivalents:

**Table 1: Delphi and HLA Integer Types**

| Delphi | HLA Equivalent | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| integer | int32[a] | -2147483648 | 2147483647 |
| cardinal | uns32[b] | 0 | 4294967295 |

3. Scalar data types are the ordinal, pointer, and real types. It does not include strings or other composite data types.

4. Delphi string objects are an exception. For reasons that have nothing to do with data representation, you should not manipulate string parameters passed in from Delphi to an HLA routine. This section will explain the problems more fully a little later.

  

**Table 1: Delphi and HLA Integer Types**

| Delphi | HLA Equivalent | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| shortint | int8 | -128 | 127 |
| smallint | int16 | -32768 | 32767 |
| longint | int32 | -2147483648 | 2147483647 |
| int64 | qword | $-2^{63}$ | $(2^{63}-1)$ |
| byte | uns8 | 0 | 255 |
| word | uns16 | 0 | 65535 |
| longword | uns32 | 0 | 4294967295 |
| subrange types | Depends on range | minimum range value | maximum range value |

a. Int32 is the implementation of integer in Delphi. Though this may change in later releases.
b. Uns32 is the implementation of cardinal in Delphi. Though this may change in later releases.

In addition to the integer values, Delphi supports several non-integer ordinal types. The following table provides their HLA equivalents:

**Table 2: Non-integer Ordinal Types in Delphi and HLA**

| Delphi | HLA | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| char | char | #0 | #255 |
| widechar | word | chr( 0 ) | chr( 65535 ) |
| boolean | boolean | false (0) | true( 1 ) |
| bytebool | byte | 0( false ) | 255 (non-zero is true) |
| wordbool | word | 0 (false ) | 65535 (non-zero is true) |
| longbool | dword | 0 (false) | 4294967295 (non-zero is true) |

  

**Table 2: Non-integer Ordinal Types in Delphi and HLA**

| Delphi | HLA | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| enumerated types | enum, byte, or word | 0 | Depends on number of items in the enumeration list. Usually the upper limit is 256 symbols |

Like the integer types, Delphi supports a wide range of real numeric formats. The following table presents these types and their HLA equivalents.

**Table 3: Real Types in Delphi and HLA**

| Delphi | HLA | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| real | real64 | 5.0 E-324 | 1.7 E+308 |
| single | real32 | 1.5 E-45 | 3.4 E+38 |
| double | real64 | 5.0 E-324 | 1.7 E+308 |
| extended | real80 | 3.6 E-4951 | 1.1 E+4932 |
| comp | real80 | $-2^{63}+1$ | $2^{63}-1$ |
| currency | real80 | -922337203685477.5808 | 922337203685477.5807 |
| real48[a] | byte[6] | 2.9 E-39 | 1.7 E+38 |

a. real48 is an obsolete type that depends upon a software floating point library. You should never use this type in assembly code. If you do, you are responsible for writing the necessary floating point subroutines to manipulate the data.

The last scalar type of interest is the pointer type. Both HLA and Delphi use a 32-bit address to represent pointers, so these data types are completely equivalent in both languages.

## 12.3.7 Passing String Data Between Delphi and HLA Code

Delphi supports a couple of different string formats. The native string format is actually very similar to HLA's string format. A string object is a pointer that points at a zero terminated sequence of characters. In the four bytes preceding the first character of the string, Delphi stores the current dynamic length of the string (just like HLA). In the four bytes before the length, Delphi stores a reference count (unlike HLA, which stores a maximum length value in this location). Delphi uses the reference count to keep track of how many different pointers contain the address of this particular string object. Delphi will automatically free the

  

storage associated with a string object when the reference count drops to zero (this is known as garbage collection).

The Delphi string format is just close enough to HLA's to tempt you to use some HLA string functions in the HLA Standard Library. This will fail for two reasons: (1) many of the HLA Standard Library string functions check the maximum length field, so they will not work properly when they access Delphi's reference count field; (2) HLA Standard Library string functions have a habit of raising string overflow (and other) exceptions if they detect a problem (such as exceeding the maximum string length value). Remember, the HLA exception handling facility is not directly compatible with Delphi's, so you should never call any HLA code that might raise an exception.

Of course, you can always grab the source code to some HLA Standard Library string function and strip out the code that raises exceptions and checks the maximum length field (this is usually the same code that raises exceptions). However, you could still run into problems if you attempt to manipulate some Delphi string. In general, it's okay to read the data from a string parameter that Delphi passes to your assembly code, but you should never change the value of such a string. To understand the problem, consider the following HLA code sequence:

```
static
    s:string := "Hello World";
    sref:string;
    scopy:string;
        .
        .
        .
    str.a_cpy( s, scopy );  // scopy has its own copy of "Hello World"

    mov( s, eax );          // After this sequence, s and sref point at
    mov( eax, sref );       // the same character string in memory.
```

After the code sequence above, any change you would make to the *scopy* string would affect only *scopy* because it has its own copy of the "Hello World" string. On the other hand, if you make any changes to the characters that *s* points at, you'll also be changing the string that *sref* points at because *sref* contains the same pointer value as *s*; in other words, *s* and *sref* are aliases of the same data. Although this aliasing process can lead to the creation of some killer defects in your code, there is a big advantage to using copy by reference rather than copy by value: copy by reference is much quicker since it only involves copying a single four-byte pointer. If you rarely change a string variable after you assign one string to that variable, copy by reference can be very efficient.

Of course, what happens if you use copy by reference to copy *s* to sref and then you want to modify the string that *sref* points at without changing the string that *s* points at? One way to do this is to make a copy of the string at the time you want to change *sref* and then modify the copy. This is known as *copy on write semantics*. In the average program, copy on write tends to produce faster running programs because the typical program tends to assign one string to another without modification more often that it assigns a string value and then modifies it later. Of course, the real problem is "how do you know whether multiple string variables are pointing at the same string in memory?" After all, if only one string variable is pointing at the string data, you don't have to make a copy of the data, you can manipulate the string data directly. The *reference counter field* that Delphi attaches to the string data solves this problem. Each time a Delphi program assigns one string variable to another, the Delphi code simply copies a pointer and then increments the reference counter. Similarly, if you assign a string address to some Delphi string variable and that variable was previously pointing at some other string data, Delphi decrements the reference counter field of that previous string value. When the reference count hits zero, Delphi automatically deallocates storage for the string (this is the garbage collection operation).

Note that Delphi strings don't need a maximum length field because Delphi dynamically allocates (standard) strings whenever you create a new string. Hence, string overflow doesn't occur and there is no need to check for string overflow (and, therefore, no need for the maximum length field). For literal string constants (which the compiler allocates statically, not dynamically on the heap), Delphi uses a reference count field of -1 so that the compiler will not attempt to deallocate the static object.

  

It wouldn't be that hard to take the HLA Standard Library strings module and modify it to use Delphi's dynamically allocated string format. There is, however, one problem with this approach: Borland has not published the internal string format for Delphi strings (the information appearing above is the result of sleuthing through memory with a debugger). They have probably withheld this information because they want the ability to change the internal representation of their string data type without breaking existing Delphi programs. So if you poke around in memory and modify Delphi string data (or allocate or deallocate these strings on your own), don't be surprised if your program malfunctions when a later version of Delphi appears (indeed, this information may already be obsolete).

Like HLA strings, a Delphi string is a pointer that happens to contain the address of the first character of a zero terminated string in memory. As long as you don't modify this pointer, you don't modify any of the characters in that string, and you don't attempt to access any bytes before the first character of the string or after the zero terminating byte, you can safely access the string data in your HLA programs. Just remember that you cannot use any Standard Library routines that check the maximum string length or raise any exceptions. If you need the length of a Delphi string that you pass as a parameter to an HLA procedure, it would be wise to use the Delphi *Length* function to compute the length and pass this value as an additional parameter to your procedure. This will keep your code working should Borland ever decide to change their internal string representation.

Delphi also supports a *ShortString* data type. This data type provides backwards compatibility with older versions of Borland's Turbo Pascal (Borland Object Pascal) product. *ShortString* objects are traditional length-prefixed strings (see "Character Strings" on page 419). A short string variable is a sequence of one to 256 bytes where the first byte contains the current dynamic string length (a value in the range 0..255) and the following *n* bytes hold the actual characters in the string (*n* being the value found in the first byte of the string data). If you need to manipulate the value of a string variable within an assembly language module, you should pass that parameter as a *ShortString* variable (assuming, of course, that you don't need to handle strings longer than 256 characters). For efficiency reasons, you should always pass *ShortString* variables by reference (or CONST or OUT) rather than by value. If you pass a short string by value, Delphi must copy all the characters allocated for that string (even if the current length is shorter) into the procedure's activation record. This can be very slow. If you pass a *ShortString* by reference, then Delphi will only need to pass a pointer to the string's data; this is very efficient.

Note that *ShortString* objects do not have a zero terminating byte following the string data. Therefore, your assembly code should use the length prefix byte to determine the end of the string, it should not search for a zero byte in the string.

If you need the maximum length of a *ShortString* object, you can use the Delphi *high* function to obtain this information and pass it to your HLA code as another parameter. Note that the high function is an compiler intrinsic much like HLA's @size function. Delphi simply replaces this "function" with the equivalent constant at compile-time; this isn't a true function you can call. This maximum size information is not available at run-time (unless you've used the Delphi *high* function) and you cannot compute this information within your HLA code.

## 12.3.8 Passing Record Data Between HLA and Delphi

Records in HLA are (mostly) compatible with Delphi records. Syntactically their declarations are very similar and if you've specified the correct Delphi compiler options you can easily translate a Delphi record to an HLA record. In this section we'll explore how to do this and learn about the incompatibilities that exist between HLA records and Delphi records.

For the most part, translating Delphi records to HLA is a no brainer. The two record declarations are so similar syntactically that conversion is trivial. The only time you really run into a problem in the conversion process is when you encounter case variant records in Delphi; fortunately, these don't occur very often and when they do, HLA's anonymous unions within a record come to the rescue.

Consider the following Pascal record type declaration:

```
type
    recType =
```

  

```
        record

            day: byte;
            month:byte;
            year:integer;
            dayOfWeek:byte;

        end;
```

The translation to an HLA record is, for the most part, very straight-forward. Just translate the field types accordingly and use the HLA record syntax (see “Records” on page 483) and you’re in business. The translation is the following:

```
type
    recType:
        record

            day: byte;
            month: byte;
            year:int32;
            dayOfWeek:byte;

        endrecord;
```

There is one minor problem with this example: data alignment. By default Delphi aligns each field of a record on the size of that object and pads the entire record so its size is an even multiple of the largest (scalar) object in the record. This means that the Delphi declaration above is really equivalent to the following HLA declaration:

```
type
    recType:
        record

            day: byte;
            month: byte;
            padding:byte[2];      // Align year on a four-byte boundary.
            year:int32;
            dayOfWeek:byte;
            morePadding: byte[3]; // Make record an even multiple of four bytes.

        endrecord;
```

Of course, a better solution is to use HLA’s ALIGN directive to automatically align the fields in the record:

```
type
    recType:
        record

            day: byte;
            month: byte;
            align( 4 );      // Align year on a four-byte boundary.
            year:int32;
            dayOfWeek:byte;
            align(4);        // Make record an even multiple of four bytes.

        endrecord;
```

Alignment of the fields is good insofar as access to the fields is faster if they are aligned appropriately. However, aligning records in this fashion does consume extra space (five bytes in the examples above) and that can be expensive if you have a large array of records whose fields need padding for alignment.

  

The alignment parameters for an HLA record should be the following:

**Table 4: Alignment of Record Fields**

| Data Type | Alignment |
|---|---|
| Ordinal Types | Size of the type: 1, 2, or 4 bytes. |
| Real Types | 2 for real48 and extended, 4 bytes for other real types |
| ShortString | 1 |
| Arrays | Same as the element size |
| Records | Same as the largest alignment of all the fields. |
| Sets | 1 or two if the set has fewer than 8 or 16 elements, 4 otherwise |
| All other types | 4 |

Another possibility is to tell Delphi not to align the fields in the record. There are two ways to do this: use the **packed** reserved word or use the {$A-} compiler directive.

The packed keyword tells Delphi not to add padding to a specific record. For example, you could declare the original Delphi record as follows:

```
type
    recType =
        packed record

            day: byte;
            month:byte;
            year:integer;
            dayOfWeek:byte;

        end;
```

With the **packed** reserved word present, Delphi does not add any padding to the fields in the record. The corresponding HLA code would be the original record declaration above, e.g.,

```
type
    recType:
        record

            day: byte;
            month: byte;
            year:int32;
            dayOfWeek:byte;

        endrecord;
```

The nice thing about the **packed** keyword is that it lets you explicitly state whether you want data alignment/padding in a record. On the other hand, if you've got a lot of records and you don't want field alignment on any of them, you'll probably want to use the "{$A-}" (turn data alignment off) option rather than

  

add the **packed** reserved word to each record definition. Note that you can turn data alignment back on with the "{$A+"} directive if you want a sequence of records to be packed and the rest of them to be aligned.

While it's far easier (and syntactically safer) to used packed records when passing record data between assembly language and Delphi, you will have to determine on a case-by-case basis whether you're willing to give up the performance gain in exchange for using less memory (and a simpler interface). It is certainly the case that packed records are easier to maintain in HLA than aligned records (since you don't have to carefully place ALIGN directives throughout the record in the HLA code). Furthermore, on new x86 processors most mis-aligned data accesses aren't particularly expensive (the cache takes care of this). However, if performance really matters you will have to measure the performance of your program and determine the cost of using packed records.

Case variant records in Delphi let you add mutually exclusive fields to a record with an optional tag field. Here are two examples:

```
type
    r1=
        record

            stdField: integer;
            case choice:boolean of
                true:( i:integer );
                false:( r:real );
        end;

    r2=
        record
            s2:real;
            case boolean of // Notice no tag object here.
                true:( s:string );
                false:( c:char );
        end;
```

HLA does not support the case variant syntax, but it does support anonymous unions in a record that let you achieve the same semantics. The two examples above, converted to HLA (assuming "{A-}") are

```
type
    r1:
        record

            stdField: int32;
            choice: boolean;   // Notice that the tag field is just another field
            union

                i:int32;
                r:real64;

            endunion;

        endrecord;

    r2:
        record

            s2:real64;
            union

                s: string;
                c: char;

            endunion;
```

  

```
    endrecord;
```

Again, you should insert appropriate ALIGN directives if you're not creating a packed record. Note that you shouldn't place any ALIGN directives inside the anonymous union section; instead, place a single ALIGN directive before the UNION reserved word that specifies the size of the largest (scalar) object in the union as given by the table "Alignment of Record Fields" on page 1179.

In general, if the size of a record exceeds about 16-32 bytes, you should pass the record by reference rather than by value.

## 12.3.9 Passing Set Data Between Delphi and HLA

Sets in Delphi can have between 1 and 256 elements. Delphi implements sets using an array of bits, exactly as HLA implements character sets (see "Character Sets" on page 441). Delphi reserves one to 32 bytes for each set; the size of the set (in bytes) is (Number_of_elements + 7) div 8. Like HLA's character sets, Delphi uses a set bit to indicate that a particular object is a member of the set and a zero bit indicates absence from the set. You can use the bit test (and set/complement/reset) instructions and all the other bit manipulation operations to manipulate character sets. Furthermore, the MMX instructions might provide a little added performance boost to your set operations (see "The MMX Instruction Set" on page 1113). For more details on the possibilities, consult the Delphi documentation and the chapters on character sets and the MMX instructions in this text.

Generally, sets are sufficiently short (maximum of 32 bytes) that passing the by value isn't totally horrible. However, you will get slightly better performance if you pass larger sets by reference. Note that HLA often passes character sets by value (16 bytes per set) to various Standard Library routines, so don't be totally afraid of passing sets by value.

## 12.3.10 Passing Array Data Between HLA and Delphi

Passing array data between some procedures written in Delphi and HLA is little different than passing array data between two HLA procedures. Generally, if the arrays are large, you'll want to pass the arrays by reference rather than value. Other than that, you should declare an appropriate array type in HLA to match the type you're passing in from Delphi and have at it. The following code fragments provide a simple example:

```
type
    PascalArray = array[0..127, 0..3] of integer;

procedure PassedArrray( var ary: PascalArray ); external;
```

Corresponding HLA code:

```
type
    PascalArray: int32[ 128, 4];

procedure PassedArray( var ary: PascalArray ); external;
```

As the above examples demonstrate, Delphi's array declarations specify the starting and ending indicies while HLA's array bounds specify the number of elements for each dimension. Other than this difference, however, you can see that the two declarations are very similar.

Delphi uses row-major ordering for arrays. So if you're accessing elements of a Delphi multi-dimensional array in HLA code, be sure to use the row-major order computation (see "Row Major Ordering" on page 469).

  

### 12.3.11 Delphi Limitations When Linking with (Non-TASM) Assembly Code

Delphi places a couple of restrictions on OBJ files that it links with the Pascal code. Some of these restrictions appears to be defects in the implementation of the linker, but only Borland can say for sure if these are defects or they are design deficiencies. The bottom line is that Delphi seems to work okay with the OBJ files that TASM produces, but fails miserably with OBJ files that other assemblers (including MASM) produce. While there are workarounds for those who insist on using the other assemblers, the only reasonable solution is to use the TASM assembler when assembling HLA output.

Note that TASM v5.0 does not support Pentium+ instructions. Further, the latest (and probably last) version of TASM (v5.3) does not support many of the newer SSE instructions. Therefore, you should avoid using these instructions in your HLA programs when linking with Delphi code.

### 12.3.12 Referencing Delphi Objects from HLA Code

Symbols you declare in the INTERFACE section of a Delphi program are public. Therefore, you can access these objects from HLA code if you declare those objects as external in the HLA program. The following sample program demonstrates this fact by declaring a structured constant (*y*) and a function (*callme*) that the HLA code uses when you press the button on a form. The HLA code calls the *callme* function (which returns the value 10) and then the HLA code stores the function return result into the *y* structured constant (which is really just a static variable).

## 12.4 Programming in C/C++ and HLA

Without question, the most popular language used to develop Win32 applications is, uh, Visual Basic. We're not going to worry about interfacing Visual Basic to assembly in this text for two reasons: (1) Visual Basic programmers will get better control and performance from their code if they learn Delphi, and (2) Visual Basic's interface to assembly is very similar to Pascal's (Delphi's) so teaching the interface to Visual Basic would repeat a lot of the material from the previous section. Coming in second as the Win32 development language of choice is C/C++. The C/C++ interface to assembly language is a bit different than Pascal/Delphi. That's why this section appears in this text.

Unlike Delphi, that has only a single vendor, there are many different C/C++ compilers available on the market. Each vendor (Microsoft, Borland, Watcom, GNU, etc.) has their own ideas about how C/C++ should interface to external code. Many vendors have their own extensions to the C/C++ language to aid in the interface to assembly and other languages. For example, Borland provides a special keyword to let Borland C++ (and C++ Builder) programmers call Pascal code (or, conversely, allow Pascal code to call the C/C++ code). Microsoft, who stopped making Pascal compilers years ago, no longer supports this option. This is unfortunate since HLA uses the Pascal calling conventions. Fortunately, HLA provides a special interface to code that C/C++ systems generate.

Before we get started discussing how to write HLA modules for your C/C++ programs, you must understand two very important facts:

> HLA's exception handling facilities are not directly compatible with C/C++'s exception handling facilities. This means that you cannot use the TRY..ENDTRY and RAISE statements in the HLA code you intend to link to a C/C++ program. This also means that you cannot call library functions that contain such statements. Since the HLA Standard Library modules use exception handling statements all over the place, this effectively prevents you from calling HLA Standard Library routines from the code you intend to link with C/C++[5].

  

Although you can write console applications with C/C++, a good percentage of C/C++ (and nearly all C++ Builder) applications are Windows/GUI applications. You cannot call console-related functions (e.g., stdin.xxxx or stdout.xxxx) from a GUI application. Even if HLA's console and standard input/output routines didn't use exception handling, you wouldn't be able to call them from a standard C/C++ application. Even if you are writing a console application in C/C++, you still shouldn't call the *stdin.xxxx* or *stdout.xxx* routines because they use the RAISE statement.

Given the rich set of language features that C/C++ supports, it should come as no surprise that the interface between the C/C++ language and assembly language is somewhat complex. Fortunately there are two facts that reduce this problem. First, HLA (v1.26 and later) supports C/C++'s calling conventions. Second, the other complex stuff you won't use very often, so you may not have to bother with it.

**Note**: the following sections assume you are already familiar with C/C++ programming. They make no attempt to explain C/C++ syntax or features other than as needed to explain the C/C++ assembly language interface. If you're not familiar with C/C++, you will probably want to skip this section.

**Also note**: although this text uses the generic term "C/C++" when describing the interface between HLA and various C/C++ compilers, the truth is that you're really interfacing HLA with the C language. There is a fairly standardized interface between C and assembly language that most vendors follow. No such standard exists for the C++ language and every vendor, if they even support an interface between C++ and assembly, uses a different scheme. In this text we will stick to interfacing HLA with the C language. Fortunately, all popular C++ compilers support the C interface to assembly, so this isn't much of a problem.

The examples in this text will use the Borland C++ compiler and Microsoft's Visual C++ compiler. There may be some minor adjustments you need to make if you're using some other C/C++ compiler; please see the vendor's documentation for more details. This text will note differences between Borland's and Microsoft's offerings, as necessary.

## 12.4.1 Linking HLA Modules With C/C++ Programs

One big advantage of C/C++ over Delphi is that (most) C/C++ compiler vendors' products emit standard object files. Well, almost standard. You wouldn't, for example, want to attempt to link the output of Microsoft's Visual C++ with TLINK (Borland's Turbo Linker) nor would you want to link the output of Borland C++ with Microsoft's linker. So, working with object files and a true linker is much nicer than having to deal with Delphi's built-in linker. As nice as the Delphi system is, interfacing with assembly language is much easier in C/C++ than in Delphi.

**Note**: the HLA Standard Library was created using Microsoft tools. This means that you will probably not be able to link this library module using the Borland TLINK program. Of course, you probably shouldn't be linking Standard Library code with C/C++ code anyway, so this shouldn't matter. However, if you really want to link some module from the HLA Standard Library with Borland C/C++, you should recompile the module and use the OBJ file directly rather than attempt to link the HLALIB.LIB file.

The Visual C++ compiler works with COFF object files. The Borland C++ compiler works with OMF object files. Both forms of object files use the OBJ extension, so you can't really tell by looking at a directory listing which form you've got. Fortunately, if you need to create a single OBJ file that will work with both, the Visual C++ compiler will also accept OMF files and convert them to a COFF file during the link phase. Of course, most of the time you will not be using both compilers, so you can pick whichever OBJ file format you're comfortable with and use that.

5. Note that the HLA Standard Library source code is available; feel free to modify the routines you want to use and remove any exception handling statements contained therein.

  

By default, HLA tells MASM to produce a COFF file when assembling the HLA output. This means that if you compile and HLA program using a command line like the following, you will not be able to directly link the code with Borland C++ code:

```
hla -c filename.hla      // The "-c" option tells HLA to compile and assemble.
```

If you want to create an OMF file rather than a COFF file, you can do so by using the following two commands:

```
hla -omf filename.hla      // The "-omf" option tells HLA to compile to OMF.
```

The execution of the above command produces and OMF object file that both VC++ and BCC (Borland C++) will accept (though VC++ prefers COFF, it accepts OMF).

Both BCC and VC++ look at the extension of the source file names you provide on the command line to determine whether they are compiling a C or a C++ program. There are some minor syntactical differences between the external declarations for a C and a C++ program. This text assumes that you are compiling C++ programs that have a ".cpp" extension. The difference between a C and a C++ compilation occurs in the external declarations for the functions you intend to write in assembly language. For example, in a C source file you would simply write:

```
extern char* RetHW( void );
```

However, in a C++ environment, you would need the following external declaration:

```
extern "C"
{
    extern char* RetHW( void );
};
```

The 'extern "C"' clause tells the compiler to use standard C linkage even though the compiler is processing a C++ source file (C++ linkage is different than C and definitely far more complex; this text will not consider pure C++ linkage since it varies so much from vendor to vendor). If you're going to compile C source files with VC++ or BCC (i.e., files with a ".c" suffix), simply drop the 'extern "C"' and the curly braces from around the external declarations.

The following sample program demonstrates this external linkage mechanism by writing a short HLA program that returns the address of a string ("Hello World") in the EAX register (like Delphi, C/C++ expects functions to return their results in EAX). The main C/C++ program then prints this string to the console device.

```
#include <stdlib.h>
#include "ratc.h"

extern "C"
{
    extern char* ReturnHW( void );
};

int main()
_begin( main )

    printf( "%s\n", ReturnHW() );
    _return 0;

_end( main )
```

Program 12.17 Cex1 - A Simple Example of a Call to an Assembly Function from C++

  

```
unit ReturnHWUnit;

    procedure ReturnHW; external( "_ReturnHW" );
    procedure ReturnHW; nodisplay; noframe; noalignstk;
    begin ReturnHW;

        lea( eax, "Hello World" );
        ret();

    end ReturnHW;

end ReturnHWUnit;
```

Program 12.18 RetHW.hla - Assembly Code that Cex1 Calls

There are several new things in both the C/C++ and HLA code that might confuse you at first glance, so let's discuss these things real quick here.

The first strange thing you will notice in the C++ code is the #include "ratc.h" statement. RatC is a C/C++ macro library that adds several new features to the C++ language. RatC adds several interesting features and capabilities to the C/C++ language, but a primary purpose of RatC is to help make C/C++ programs a little more readable. Of course, if you've never seen RatC before, you'll probably argue that it's not as readable as pure C/C++, but even someone who has never seen RatC before can figure out 80% of Ratc within a minutes. In the example above, the _begin and _end clauses clearly map to the "{" and "}" symbols (notice how the use of _begin and _end make it clear what function or statement associates with the braces; unlike the guesswork you've got in standard C). The _return statement is clearly equivalent to the C return statement. As you'll quickly see, all of the standard C control structures are improved slightly in RatC. You'll have no trouble recognizing them since they use the standard control structure names with an underscore prefix. This text promotes the creation of readable programs, hence the use of RatC in the examples appearing in this chapter[6]. You can find out more about RatC on Webster at http://webster.cs.ucr.edu.

The C/C++ program isn't the only source file to introduce something new. If you look at the HLA code you'll notice that the LEA instruction appears to be illegal. It takes the following form:

```
lea( eax, "Hello World" );
```

The LEA instruction is supposed to have a memory and a register operand. This example has a register and a constant; what is the address of a constant, anyway? Well, this is a syntactical extension that HLA provides to 80x86 assembly language. If you supply a constant instead of a memory operand to LEA, HLA will create a static (readonly) object initialized with that constant and the LEA instruction will return the address of that object. In this example, HLA will emit the string to the constants segment and then load EAX with the address of the first character of that string. Since HLA strings always have a zero terminating byte, EAX will contain the address of a zero-terminated string which is exactly what C++ wants. If you look back at the original C++ code, you will see that *RetHW* returns a *char** object and the main C++ program displays this result on the console device.

If you haven't figured it out yet, this is a round-about version of the venerable "Hello World" program.

Microsoft VC++ users can compile this program from the command line by using the following commands[7]:

```
hla -c RetHW.hla          // Compiles and assembles RetHW.hla to RetHW.obj
```

6. If RatC really annoys you, just keep in mind that you've only got to look at a few RatC programs in this chapter. Then you can go back to the old-fashioned C code and hack to your heart's content!

7. This text assumes you've executed the VCVARS32.BAT file that sets up the system to allow the use of VC++ from the command line.

  

```
cl Cex1.cpp RetHW.obj     // Compiles C++ code and links it with RetHW.obj
```

If you're a Borland C++ user, you'd use the following command sequence:

```
hla -o:omf RetHW.hla        // Compile HLA file to an OMF file.
bcc32i Cex1.cpp RetHW.obj   // Compile and link C++ and assembly code.
                            // Could also use the BCC32 compiler.
```

GCC users can compile this program from the command line by using the following commands:

```
hla -o:omf RetHW.hla        // Compile HLA file to an OMF file.
bcc32i Cex1.cpp RetHW.obj   // Compile and link C++ and assembly code.
                            // Could also use the BCC32 compiler.
```

## 12.4.2 Register Preservation

Unlike Delphi, a single language with a single vendor, there is no single list of registers that you can freely use as scratchpad values within an assembly language function. The list changes by vendor and even changes between versions from the same vendor. However, you can safely assume that EAX is available for scratchpad use since C functions return their result in the EAX register. You should probably preserve everything else.

## 12.4.3 Function Results

C/C++ compilers universally seem to return ordinal and pointer function results in AL, AX, or EAX depending on the operand's size. The compilers probably return floating point results on the top of the FPU stack as well. Other than that, check your C/C++ vendor's documentation for more details on function return locations.

## 12.4.4 Calling Conventions

The standard C/C++ calling convention is probably the biggest area of contention between the C/C++ and HLA languages. VC++ and BCC both support multiple calling conventions. BCC even supports the Pascal calling convention that HLA uses, making it trivial to write HLA functions for BCC programs[8]. However, before we get into the details of these other calling conventions, it's probably a wise idea to first discuss the standard C/C++ calling convention.

Both VC++ and BCC *decorate* the function name when you declare an external function. For external "C" functions, the decoration consists of an underscore. If you look back at Program 12.18 you'll notice that the external name the HLA program actually uses is "_RetHW" rather than simply "RetHW". The HLA program itself, of course, uses the symbol "RetHW" to refer to the function, but the external name (as specified by the optional parameter to the EXTERNAL option) is "_RetHW". In the C/C++ program (Program 12.17) there is no explicit indication of this decoration; you simply have to read the compiler documentation to discover that the compiler automatically prepends this character to the function name[9]. Fortunately, HLA's EXTERNAL option syntax allows us to *undecorate* the name, so we can refer to the function using the same name as the C/C++ program. Name decoration is a trivial matter, easily fixed by HLA.

A big problem is the fact that C/C++ pushes parameters on the stack in the opposite direction of just about every other (non-C based) language on the planet; specifically, C/C++ pushes actual parameters on

8. Microsoft used to support the Pascal calling convention, but when they stopped supporting their QuickPascal language, they dropped support for this option.

9. Most compilers provide an option to turn this off if you don't want this to occur. We will assume that this option is active in this text since that's the standard for external C names.

  

the stack from right to left instead of the more common left to right. This means that you cannot declare a C/C++ function with two or more parameters and use a simple translation of the C/C++ external declaration as your HLA procedure declaration, i.e., the following are not equivalent:

```
external void CToHLA( int p, unsigned q, double r );
procedure CToHLA( p:int32; q:uns32; r:real64 ); external( "_CToHLA" );
```

Were you to call *CToHLA* from the C/C++ program, the compiler would push the *r* parameter first, the *q* parameter second, and the *p* parameter third - exactly the opposite order that the HLA code expects. As a result, the HLA code would use the L.O. double word of *r* as *p's* value, the H.O. double word of *r* as *q's* value, and the combination of *p* and *q's* values as the value for *r*. Obviously, you'd most likely get an incorrect result from this calculation. Fortunately, there's an easy solution to this problem: use the @CDECL procedure option in the HLA code to tell it to reverse the parameters:

```
procedure CToHLA( p:int32; q:uns32; r:real64 ); @cdecl; external( "_CToHLA" );
```

Now when the C/C++ code calls this procedure, it push the parameters on the stack and the HLA code will retrieve them in the proper order.

There is another big difference between the C/C++ calling convention and HLA: HLA procedures automatically clean up after themselves by removing all parameters pass to a procedure prior to returning to the caller. C/C++, on the other hand, requires the caller, not the procedure, to clean up the parameters. This has two important ramifications: (1) if you call a C/C++ function (or one that uses the C/C++ calling sequence), then your code has to remove any parameters it pushed upon return from that function; (2) your HLA code cannot automatically remove parameter data from the stack if C/C++ code calls it. The @CDECL procedure option tells HLA not to generate the code that automatically removes parameters from the stack upon return. Of course, if you use the @NOFRAME option, you must ensure that you don't remove these parameters yourself when your procedures return to their caller.

One thing HLA cannot handle automatically for you is removing parameters from the stack when you call a procedure or function that uses the @CDECL calling convention; for example, you must manually pop these parameters whenever you call a C/C++ function from your HLA code.

Removing parameters from the stack when a C/C++ function returns to your code is very easy, just execute an "add( constant, esp );" instruction where *constant* is the number of parameter bytes you've pushed on the stack. For example, the *CToHLA* function has 16 bytes of parameters (two *int32* objects and one *real64* object) so the calling sequence (in HLA) would look something like the following:

```
CToHLA( pVal, qVal, rVal );  // Assume this is the macro version.
add( 16, esp );              // Remove parameters from the stack.
```

Cleaning up after a call is easy enough. However, if you're writing the function that must leave it up to the caller to remove the parameters from the stack, then you've got a tiny problem – by default, HLA procedures always clean up after themselves. If you use the @CDECL option and don't specify the @NOFRAME option, then HLA automatically handles this for you. However, if you use the @NOFRAME option, then you've got to ensure that you leave the parameter data on the stack when returning from a function/procedure that uses the @CDECL calling convention.

If you want to leave the parameters on the stack for the caller to remove, then you must write the standard entry and exit sequences for the procedure that build and destroy the activation record (see "The Standard Entry Sequence" on page 813 and "The Standard Exit Sequence" on page 814). This means you've got to use the @NOFRAME (and @NODISPLAY) options on your procedures that C/C++ will call. Here's a sample implementation of the CToHLA procedure that builds and destroys the activation record:

```
procedure _CToHLA( rValue:real64; q:uns32; p:int32 ); @nodisplay; @noframe;
begin _CToHLA;

    push( ebp );             // Standard Entry Sequence
    mov( esp, ebp );
    // sub( _vars_, esp );   // Needed if you have local variables.
        .
        .       // Code to implement the function's body.
```

  

```
        .
    mov( ebp, esp );        // Restore the stack pointer.
    pop( ebp );             // Restore link to previous activation record.
    ret();                  // Note that we don't remove any parameters.

end _CToHLA;
```

If you're willing to use some vendor extensions to the C/C++ programming language, then you can make the interface to HLA much simpler. For example, if you're using Borland's C++ product, it has an option you can apply to function declarations to tell the compiler to use the Pascal calling convention. Since HLA uses the Pascal calling convention, specifying this option in your BCC programs will make the interface to HLA trivial. In Borland C++ you can specify the Pascal calling convention for an external function using the following syntax:

```
extern type  _pascal  funcname( parameters )
```

Example:

```
extern void _pascal CToHLA( int p, unsigned q, double r );
```

The Pascal calling convention does not decorate the name, so the HLA name would not have a leading underscore. The Pascal calling convention uses case insensitive names; BCC achieves this by converting the name to all uppercase. Therefore, you'd probably want to use an HLA declaration like the following:

```
procedure CToHLA( p:int32; q:uns32; r:real64 ); external( "CTOHLA" );
```

Procedures using the Pascal calling convention push their parameters from left to right and leave it up to the procedure to clean up the stack upon return; exactly what HLA does by default. When using the Pascal calling convention, you could write the *CToHLA* function as follows:

```
procedure CToHLA( rValue:real64; q:uns32; p:int32 ); external( "CTOHLA" );

procedure CToHLA( rValue:real64; q:uns32; p:int32 ); nodisplay; noalignstk;
begin CToHLA;
        .
        .       // Code to implement the function's body.
        .
end CToHLA;
```

Note that you don't have to supply the standard entry and exit sequences. HLA provides those automatically.

Of course, Microsoft isn't about to support the Pascal calling sequence since they don't have a Pascal compiler. So this option isn't available to VC++ users.

Both Borland and Microsoft (and HLA) support the so-called *StdCall* calling convention. This is the calling convention that Windows uses, so nearly every language that operates under Windows provides this calling convention. The StdCall calling convention is a combination of the C and Pascal calling conventions. Like C, the functions need to have their parameters pushed on the stack in a right to left order; like Pascal, it is the caller's responsibility to clean up the parameters when the function returns; like C, the function name is case sensitive; like Pascal, the function name is not decorated (i.e., the external name is the same as the function declaration). The syntax for a StdCall function is the same in both VC++ and BCC, it is the following:

```
extern void _stdcall CToHLA( int p, unsigned q, double r );
```

HLA supports the StdCall convention using the STDCALL procedure option.. Because the name is undecorated, you could use a prototype and macro like the following:

```
procedure CToHLA( p:int32; q:uns32; r:real64 ); stdcall; external( "CToHLA" );
procedure CToHLA( p:int32; q:uns32; r:real64  );  nodisplay; nostkalign;
begin CToHLA;
    .
```

  

```
    .  // Function body
    .
end CToHLA;
    .
    .
    .
CToHLA( pValue, qValue, rValue );  // Demo of a call to CToHLA.
```

## 12.4.5 Pass by Value and Reference in C/C++

A C/C++ program can pass parameters to a procedure or function using one of two different mechanisms: pass by value and pass by reference. Since pass by reference parameters use pointers, this parameter passing mechanism is completely compatible between HLA and C/C++. The following two lines provide an external declaration in C++ and the corresponding external (public) declaration in HLA for a pass by reference parameter using the calling convention:

```
extern void HasRefParm( int& refparm );                          // C++
procedure HasRefParm( var refparm: int32 ); external;            // HLA
```

Like HLA, C++ will pass the 32-bit address of whatever actual parameter you specify when calling the *HasRefParm* procedure. Don't forget, inside the HLA code, that you must dereference this pointer to access the actual parameter data. See the chapter on Intermediate Procedures for more details (see "Pass by Reference" on page 817).

Like HLA, C++ lets you pass untyped parameters by reference. The syntax to achieve this in C++ is the following:

```
extern void UntypedRefParm( void* parm1 );
```

Actually, this is not a reference parameter, but a value parameter with an untyped pointer.

In HLA, you can use the VAR keyword as the data type to specify that you want an untyped reference parameter. Here's the corresponding prototype for the *UntypedRefParm* procedure in HLA:

```
procedure UntypedRefParm( var parm1:var );
external;
```

## 12.4.6 Scalar Data Type Correspondence Between C/C++ and HLA

When passing parameters between C/C++ and HLA procedures and functions, it's very important that the calling code and the called code agree on the basic data types for the parameters. In this section we will draw a correspondence between the C/C++ scalar data types and the HLA (v1.x) data types.

Assembly language supports any possible data format, so HLA's data type capabilities will always be a superset of C/C++'s. Therefore, there may be some objects you can create in HLA that have no counterpart in C/C++, but the reverse is not true. Since the assembly functions and procedures you write are generally manipulating data that C/C++ provides, you don't have to worry too much about not being able to process some data passed to an HLA procedure by C/C++.

C/C++ provides a wide range of different integer data types. Unfortunately, the exact representation of these types is implementation specific. The following table lists the C/C++ types as currently implemented by Borland C++ and Microsoft VC++. This table may very well change as 64-bit compilers become available.

  

**Table 5: C/C++ and HLA Integer Types**

| C/C++ | HLA Equivalent | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| int | int32 | -2147483648 | 2147483647 |
| unsigned | uns32 | 0 | 4294967295 |
| signed char | int8 | -128 | 127 |
| short | int16 | -32768 | 32767 |
| long | int32 | -2147483648 | 2147483647 |
| unsigned char | uns8 | 0 | 255 |
| unsigned short | uns16 | 0 | 65535 |

In addition to the integer values, C/C++ supports several non-integer ordinal types. The following table provides their HLA equivalents:

**Table 6: Non-integer Ordinal Types in C/C++ and HLA**

| C/C++ | HLA | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| wchar, TCHAR | word | 0 | 65535 |
| BOOL | boolean | false (0) | true ( not zero ) |

Like the integer types, C/C++ supports a wide range of real numeric formats. The following table presents these types and their HLA equivalents.

**Table 7: Real Types in C/C++ and HLA**

| C/C++ | HLA | Range | |
|---|---|---|---|
| | | Minimum | Maximum |
| double | real64 | 5.0 E-324 | 1.7 E+308 |
| float | real32 | 1.5 E-45 | 3.4 E+38 |
| long double[a] | real80 | 3.6 E-4951 | 1.1 E+4932 |

a. This data type is 80 bits only in BCC. VC++ uses 64 bits for the long double type.

  

The last scalar type of interest is the pointer type. Both HLA and C/C++ use a 32-bit address to represent pointers, so these data types are completely equivalent in both languages.

## 12.4.7 Passing String Data Between C/C++ and HLA Code

C/C++ uses zero terminated strings. Algorithms that manipulate zero-terminated strings are not as efficient as functions that work on length-prefixed strings; on the plus side, however, zero-terminated strings are very easy to work with. HLA's strings are downwards compatible with C/C++ strings since HLA places a zero byte at the end of each HLA string. Since you'll probably not be calling HLA Standard Library string routines, the fact that C/C++ strings are not upwards compatible with HLA strings generally won't be a problem. If you do decide to modify some of the HLA string functions so that they don't raise exceptions, you can always translate the *str.cStrToStr* function that translates zero-terminated C/C++ strings to HLA strings.

A C/C++ string variable is typically a char* object or an array of characters. In either case, C/C++ will pass the address of the first character of the string to an external procedure whenever you pass a string as a parameter. Within the procedure, you can treat the parameter as an indirect reference and dereference to pointer to access characters within the string.

## 12.4.8 Passing Record/Structure Data Between HLA and C/C++

Records in HLA are (mostly) compatible with C/C++ structs. You can easily translate a C/C++ struct to an HLA record. In this section we'll explore how to do this and learn about the incompatibilities that exist between HLA records and C/C++ structures.

For the most part, translating C/C++ records to HLA is a no brainer. Just grab the "guts" of a structure declaration and translate the declarations to HLA syntax within a RECORD..ENDRECORD block and you're done.

Consider the following C/C++ structure type declaration:

```
typedef struct
{
    unsigned char day;
    unsigned char month;
    int year;
    unsigned char dayOfWeek;
} dateType;
```

The translation to an HLA record is, for the most part, very straight-forward. Just translate the field types accordingly and use the HLA record syntax (see "Records" on page 483) and you're in business. The translation is the following:

```
type
    recType:
        record

            day: byte;
            month: byte;
            year:int32;
            dayOfWeek:byte;

        endrecord;
```

There is one minor problem with this example: data alignment. Depending on your compiler and whatever defaults it uses, C/C++ might not pack the data in the structure as compactly as possible. Some C/C++ compilers will attempt to align the fields on double word or other boundaries. With double word alignment of objects larger than a byte, the previous C/C++ **typedef** statement is probably better modelled by

  

```
type
    recType:
        record

            day: byte;
            month: byte;
            padding:byte[2];      // Align year on a four-byte boundary.
            year:int32;
            dayOfWeek:byte;
            morePadding: byte[3]; // Make record an even multiple of four bytes.

        endrecord;
```

Of course, a better solution is to use HLA's ALIGN directive to automatically align the fields in the record:

```
type
    recType:
        record

            day: byte;
            month: byte;
            align( 4 );      // Align year on a four-byte boundary.
            year:int32;
            dayOfWeek:byte;
            align(4);        // Make record an even multiple of four bytes.

        endrecord;
```

Alignment of the fields is good insofar as access to the fields is faster if they are aligned appropriately. However, aligning records in this fashion does consume extra space (five bytes in the examples above) and that can be expensive if you have a large array of records whose fields need padding for alignment.

You will need to check your compiler vendor's documentation to determine whether it packs or pads structures by default. Most compilers give you several options for packing or padding the fields on various boundaries. Padded structures might be a bit faster while packed structures (i.e., no padding) are going to be more compact. You'll have to decide which is more important to you and then adjust your HLA code accordingly.

Note that by default, C/C++ passes structures by value. A C/C++ program must explicitly take the address of a structure object and pass that address in order to simulate pass by reference. In general, if the size of a structure exceeds about 16 bytes, you should pass the structure by reference rather than by value.

## 12.4.9 Passing Array Data Between HLA and C/C++

Passing array data between some procedures written in C/C++ and HLA is little different than passing array data between two HLA procedures. First of all, C/C++ can only pass arrays by reference, never by value. Therefore, you must always use pass by reference inside the HLA code. The following code fragments provide a simple example:

```
    int CArray[128][4];

extern void PassedArrray( int array[128][4] );
```

Corresponding HLA code:

```
type
    CArray: int32[ 128, 4];

procedure PassedArray( var ary: CArray ); external;
```

  

As the above examples demonstrate, C/C++'s array declarations are similar to HLA's insofar as you specify the bounds of each dimension in the array.

C/C++ uses row-major ordering for arrays. So if you're accessing elements of a C/C++ multi-dimensional array in HLA code, be sure to use the row-major order computation (see "Row Major Ordering" on page 469).

## 12.5 Putting It All Together

Most real-world assembly code that is written consists of small modules that programmers link to programs written in other languages. Most languages provide some scheme for interfacing that language with assembly (HLA) code. Unfortunately, the number of interface mechanisms is sufficiently close to the number of language implementations to make a complete exposition of this subject impossible. In general, you will have to refer to the documentation for your particular compiler in order to learn sufficient details to successfully interface assembly with that language.

Fortunately, nagging details aside, most high level languages do share some common traits with respect to assembly language interface. Parameter passing conventions, stack clean up, register preservation, and several other important topics often apply from one language to the next. Therefore, once you learn how to interface a couple of languages to assembly, you'll quickly be able to figure out how to interface to others (given the documentation for the new language).

This chapter discusses the interface between the Delphi and C/C++ languages and assembly language. Although there are more popular languages out there (e.g., Visual Basic), Delphi and C/C++ introduce most of the concepts you'll need to know in order to interface a high level language with assembly language. Beyond that point, all you need is the documentation for your specific compiler and you'll be interfacing assembly with that language in no time.

  

 