11 HLA Procedure Declarations and Procedure Calls

Note: this chapter discusses how HLA generates code for various procedure calls and procedure bodies. The examples given here should be treated as gross approximations only. Most of these examples come from early version of HLA v1.x and later versions have substantially improved the code generation. When in doubt, compile a test program with HLA emitting source code for your favorite assembler and view the output that HLA procedures.

11.1 Procedure Declarations

HLA supports two different ways to declare procedures in a program: the "original style" ("old style") and the "new style". The "original style" was introduced in HLA v1.0; the "new style" of procedure declarations was introduced in HLA v2.0. Note that the "original style" (despite often being called the "old style") is not obsolete nor is it in danger of being deprecated. The original and new styles of procedure declarations are complementary. The new procedure declaration style simply adds some consistency and enhanced facilities to HLA.

11.1.1 Original Style Procedure Declarations

Original style procedure declarations are nearly identical to program declarations with two major differences: procedures are declared using the "procedure" reserved word and procedures may have parameters. The general syntax is:

```
procedure identifier ( optional_parameter_list ); procedure_options
   declarations
begin identifier;
   statements
end identifier;
```

Note that you may declare procedures inside other procedure in a fashion analogous to most block-structured languages (e.g., Pascal).

The optional parameter list consists of a list of **var**-type declarations taking the form:

```
optional_access_keyword  identifier1 : identifier2 optional_in_reg
```

optional_access_keyword, if present, must be **val**, **var**, **valres**, **result**, **name**, or **lazy** and defines the parameter passing mechanism (pass by value, pass by reference, pass by value/result [or value/returned], pass by result, pass by name, or pass by lazy evaluation, respectively). The default is pass by value (**val**) if an access keyword is not present. For pass by value parameters, HLA allocates the specified number of bytes according to the size of that object in the activation record. For pass by reference, pass by value/result, and pass by result, HLA allocates four bytes to hold a pointer to the object. For pass by name and pass by lazy evaluation, HLA allocates eight bytes to hold a pointer to the associated thunk and a pointer to the thunk's execution environment (see the sections on parameters and thunks for more details).

The *optional_in_reg* clause, if present, corresponds to the phrase "in *reg*" where *reg* is one of the 80x86's general-purpose 8-, 16-, or 32-bit registers. You must take care when passing parameters through the registers as the parameter names become aliases for registers and this can create confusion when reading the code later (especially if, within a procedure with a register parameter, you call another procedure that uses that same register as a parameter).

HLA also allows a special parameter of the form:

```
var identifer : var
```

This creates an untyped reference parameter. You may specify any memory variable as the corresponding actual parameter and HLA will compute the address of that object and pass it on to the procedure without further type checking. Within the procedure, the parameter is given the **dword** type.

The *procedure_options* component above is a list of keywords that specify how HLA emits code for the procedure. There are several different procedure options available: `@noalignstack`, `@alignstack`, `@pascal`, `@stdcall`, `@cdecl`, `@align`(*int_const*), `@use reg32`, @leave, @noleave, @enter, @noenter, and `@returns`("text").

Option	Description
@noframe, @frame	By default, HLA emits code at the beginning of the procedure to construct a stack frame. The `@noframe` option disables this action. The **@**`frame` option tells HLA to emit code for a particular procedure if stack frame generation is off by default. HLA also uses these two special reserved words as a compile-time variables to set the default frame generation for all procedures. Setting **@frame** to true (or **@noframe** to false) turns on frame generation by default; setting **@frame** to false (or **@noframe** to true) turns off frame generation.
@nodisplay, @display	By default, HLA emits code at the beginning of the procedure to construct a display within the frame. The `@nodisplay` option disables this action. The **@**`display` option tells HLA to emit code to generate a display for a particular procedure if display generation is off by default. Note that HLA does not emit code to construct the display if **@**`noframe` is in effect, though it will assume that the programmer will construct this display themself. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting **@display** to true (or **@nodisplay** to false) turns on display generation by default; setting **@display** to false (or **@nodisplay** to true) turns off display generation.
@noalignstack, @alignstack	By default (assuming **@frame** generation is active), HLA will emit an instruction to align ESP on a four-byte boundary after allocating local variables. Win32, *NIX, and other 32-bit OSes require the stack to be double-word-aligned (hence this option). If you know the stack will be double-word-aligned, you can eliminate this extra instruction by specifying the `@noalignstack` option. Conversely, you can force the generation of this instruction by specifying the **@**`alignstack` procedure option. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting **@alignstack** to true (or **@noalignstack** to false) turns on stack alignment generation by default; setting **@alignstack** to false (or **@noalignstack** to true) turns off stack alignment code generation.

@pascal, @cdecl, @stdcall	These options give you the ability to specify the parameter passing mechanism for procedures. By default, HLA uses the **@**`pascal` calling sequence for all procedures. This calling sequence pushes the parameters on the stack in a left-to-right order (i.e., in the order they appear in the parameter list). It also automatically cleans up the stack upon return from the procedure. The **@cdecl** procedure option tells HLA to pass the parameters from right-to-left so that the first parameter appears at the lowest address in memory and it is the user's responsibility to remove the parameters from the stack upon return from the procedure. The `@stdcall` procedure option is a hybrid of the **@**`pascal` and **@**`cdecl` calling conventions. It pushes the parameters in the right-to-left order (just like **@**`cdecl`) but **@**`stdcall` procedures automatically remove their parameter data from the stack (just like **@**`pascal`). Win32 API calls use the **@**`stdcall` calling convention. Note that iterators and methods always use the Pascal calling convention; you may only apply the **@cdecl** and **@stdcall** options to HLA procedures.
@align(int_constant)	The **@**`align`(*int_const*) procedure option aligns the procedure on a 1, 2, 4, 8, or 16 byte boundary. Specify the boundary you desire as the parameter to this option. The default is `@align(1)`, which is unaligned; HLA also uses this special identifiers as a compile-time variable to set the default procedure alignment . Setting **@align** := 1 turns off procedure alignment while supplying some other value (which must be a power of two) sets the default procedure alignment to the specified number of bytes.
@use reg_{32}	When passing parameters, HLA can sometimes generate better code if it has a 32-bit general purpose register for use as a scratchpad register. By default, HLA never modifies the value of a register behind your back; so, it will often generate less than optimal code when passing certain parameters on the stack. By using the @use procedure option, you can specify one of the following 32-bit registers for use by HLA: `EAX, EBX, ECX, EDX, ESI`, or `EDI`. By providing one of these registers, HLA may be able to generate significantly better code when passing certain parameters.
@returns("text")	This option specifies the compile-time return value whenever a function name appears as an instruction operand. For example, suppose you are writing a function that returns its result in EAX. You should probably specify a "returns" value of "EAX" so you can compose that procedure just like any other HLA machine instruction (see the example below and the section on machine instructions for more details).

@leave,
@noleave
	These two options control the code generation for the standard exit sequence. If you specify the `@leave` option then HLA emits the x86 `leave` instruction to clean up the activation record before the procedure returns. If you specify the `@noleave` option, then HLA emits the primitive instructions to achieve this, e.g.,
	mov(ebp, esp);
	pop(ebp);
	
	The manual sequence is faster on some architectures, the `leave` instruction is always shorter.
	
	Note that `@noleave` occurs by default if you've specified `@noframe`. By default, HLA assumes `@noleave` but you may change the default using these special identifiers as a compile-time variable to set the default **leave** generation for all procedures. Setting **@leave** to true (or **@noleave** to false) turns on **leave** generation by default; setting **@leave** to false (or **@noleave** to true) turns off the use of the **leave** instruction.

@enter, @noenter	These two options control the code generation for a procedure's standard entry sequence. If you specify the `@enter` option then HLA emits the x86 `enter` instruction to create the activation record. If you specify the `@noenter` option, then HLA emits the primitive instructions to achieve this. The manual sequence is always faster, using the `enter` instruction is usually shorter. Note that `@noenter` occurs by default if you've specified `@noframe`. By default, HLA assumes `@noenter` but you may change the default using these special identifiers as a compile-time variable to set the default **enter** generation for all procedures. Setting **@enter** to true (or **@noenter** to false) turns on **enter** generation by default; setting **@enter** to false (or **@noenter** to true) turns off the use of the **enter** instruction.

The following example demonstrates how the **@**`returns` option works:

```
program returnsDemo;
#include( "stdio.hhf" );

    procedure eax0; @returns( "eax" );
    begin eax0;

        mov( 0, eax );

    end eax0;

begin returnsDemo;

    mov( eax0(), ebx );
```

```
    stdout.put( "ebx=", ebx, nl );

end returnsDemo;
```

11.1.2 "New Style" Procedure Declarations

HLA v2.0 added a new form of procedure declaration to make the syntax of procedure declarations more consistent with the other declaration sections (i.e., **const, val, type, var, static, readonly,** and **storage**). The new syntax uses the **proc** keyword to begin a procedure declaration section, e.g.,

```
proc
<< procedure declarations using new style syntax>>
```

You may optionally end a **proc** section with an **endproc** clause:

```
proc
 << procedure declarations using new style syntax>>
endproc;
```

A **procedure, iterator**, or **method** declaration appearing in a **proc** section is declared using one of the following forms:

```
identifier : procedure_type;
begin identifier;
   <<procedure body>>
end identifier;

identifier : procedure_type  procedure_options;
begin identifier;
   <<procedure body>>
end identifier;
```

identifier is the name of the procedure you're declaring. This is a standard HLA identifier.

procedure_type is either a predefined (in the type section) procedure type or the reserved word **procedure** followed by an optional parameter list. Here are some examples of procedure declarations using both schemes:

```
type
   proc_t : procedure( i:int32; u:uns32 );

proc
   proc1:proc_t;
   begin proc1;
      <<procedure body>>
   end proc1;

   proc2:procedure( a:char );
   begin proc2;
      <<procedure body>>
   end proc2;
endproc;
```

The advantage of using a procedure type identifier to capture the parameter list is especially evident when defining several **forward** or **external** procedure declarations in a header file. If you have several procedures that all have the same parameter list (e.g., Win32 *winproc* type procedures), you can save a lot of typing by specifying a generic procedure type rather than repeating the same parameter list over and over again in the procedure declarations (using the original style declarations).

The syntax for procedure options in the new style procedure declarations is slightly different from the original style. Procedure options, if present, appear after the procedure type (or parameter list) and are surrounded by a pair of braces; they are separated by spaces or commas rather than terminated by semicolons. Here are some examples:

```
type
   proc_t : procedure( i:int32; u:uns32);

proc
   procx:proc_t; external;
   procy:procedure( d:dword ) { @returns( "@c" )}; forward;

   proc1 :proc_t {@noframe, @nodisplay};
   begin proc1;
      <<procedure body>>
   end proc1;

   proc2:procedure( var a:char )
      {@cdecl, @use eax, @returns( "eax" ), @noframe};
   begin proc2;
      <<procedure body>>
   end proc2;
```

Note that an original style **procedure** (or **iterator** or **method**) declaration will terminate a **proc** section. If you want to add some additional new style procedure declarations after an original style declaration, you will have to begin a new **proc** section by supplying another **proc** keyword definition.

HLA v1.x provided the ability to create a pointer to a procedure using syntax like the following:

```
procedure someProc( i:int32; u:uns32);
begin someProc;
   << procedure body >>
end someProc;
   .
   .
   .
type
   someProc_t :pointer to someProc;

static
   ptrToSomeProc:someProc_t;
```

Objects of *someProc_t* will be pointers to procedures have two parameters (**int32** and **uns32**, as per the original *someProc* declaration). This syntax has always been a kludge (it was added at the request of an early HLA user) but the original HLA procedure declaration syntax really didn't allow anything that was better. Basing a type on an instance of an existing procedure declaration is ugly syntax. The better solution is to do the converse, base the procedure declaration on some procedure type. This is what the new procedure declaration syntax does.

```
type
    someProc_t :procedure( i:int32; u:uns32);

proc
    someProc :someProc_t;
    begin someProc;
        << procedure body >>
    end someProc;
        .
        .
        .

static
    ptrToSomeProc:someProc_t;
```

11.2 Overloaded Procedure/Iterator/Method Declarations

Starting with HLA v2.5, the HLA language supports an **overloads** declaration in the **proc** declaration section. In previous versions of HLA it was possible to use the **overload** macro to create overloaded procedures, but that macro has some serious limitations. The new **overloads** declaration lifts many of the restrictions of the overload macro.

Because the **overloads** declaration replaces the functionality of the **overload** macro, the macro was moved out of the hla.hhf header file and into its own overload.hhf header file. The stdlib.hhf header file does not automatically include overload.hhf; therefore, if you are using the **overload** macro in your programs you will need to explicitly include the overload.hhf header file to compile without error. Better yet, convert your existing code to use the new **overloads** declaration.

Overloaded functions allows you to call a function (procedure, method, or iterator) based on a calling *signature* rather than just by the function's name. A call signature consists of the function's name and the number and types of its parameter list. Several functions can share the same name but have different signatures based on their parameter lists. For example, consider a **put** function with the following signatures:

```
procedure put( u:ins32 );
procedure put( i:int32 );
procedure put( c:char );
```

If you call **put** with an **uns32** parameter, it will invoke the version of the procedure that has an **uns32** argument; likewise, if you pass an **int32** or **char** parameter, the call to **put** will invoke the corresponding procedure.

The number of parameters also affects the function's signature; consider an extension of the **put** procedure signatures:

```
procedure put( u:ins32 );
procedure put( u:ins32; size:int32 );
procedure put( i:int32 );
procedure put( i:int32; size:int32 );
procedure put( c:char );
procedure put( c:char; size:int32 );
```

An invocation of the form "put(u32Var);" will call the first version of **put**; an invocation of the form "put(u32Var, width);" will call the second version above (assuming *u32Var* is an **uns32** object and *width* is an **int32** object).

Unlike some HLLs, HLA doesn't allow you to declare multiple procedures with the same name (but different signatures). The problem with that approach is that HLA has to generated internal ("mangled") names and this creates problems when you try to link overloaded procedures with other code (particularly if it isn't written in that same language). Instead, HLA's overloads declaration takes a more reasonable approach where you define the names of the individual functions (using standard procedure declarations with unique function names) and then use a set of overloads declarations to assign different signatures (with the same name) to these functions.

The **overloads** declaration appears in a **proc** section and uses the following basic syntax:

```
ovldName : overloads functionName;
ovldName : overloads functionName ( "string calling sequence" );
```

Here are a couple of overload example declarations:

```
procedure proc1( i:int32 ); external;
procedure proc2( i:int32; j:int32); external;
procedure proc3( i:in32; j:int32; k:int32); external;

proc
    prc : overloads proc1;
    prc : overloads proc2;
    prc : overloads proc3;
```

In this example, *prc* is the overloaded procedure name. Invoking *prc* will call *proc1, proc2,* or *proc3*, depending on the complete invocation signature. Note that you don't actually create a procedure named prc. The **overloads** declaration tells HLA that *ovldName* overloads the *functionName* in the overloads declaration.

In the **overloads** declaration, *functionName* must be the name of an already-defined **procedure**, **method**, or **iterator**. This can be a **forward**, **external**, or actual **procedure/iterator/ method** declaration. Note that you cannot specify a procedure type name here; only actual **procedure/method/iterator** names are legal (as in the example above).

The overloaded name (e.g., *prc* in the example above) must be either an undefined identifier or an existing **overloads** name. **Overloads** identifiers follow all the usual HLA scoping rules with one extension: if you declare an **overloads** identifier inside a **procedure, method,** or **iterator** and that identifier is an overloads identifier outside that function, then the local declaration will extend the global declaration, not replace it. For example, consider this code:

```
program example;
    static
        i:int32;
        u:int32;

    proc
        proc1 :procedure( i:int32 ); external;
        proc2 :procedure ( u:uns32 ); external;

        prc : overloads proc1;

    procedure local;
    proc
        prc :overloads proc2;
    begin local;

        prc( i ); // Calls proc1
        prc( u ); // Calls proc2

    end local;
```

```
begin example;

   prc( i ); // calls proc1
   prc( u ); // Illegal, no valid signature for this call

end example;
```

Note how the signature that has an **uns32** argument is visible only inside *local*. Also note that the *prc* signature defined outside *local* is also usable inside *local*, despite the local declaration of *prc* inside the *local* procedure. Outside the *local* procedure, *prc* is still valid but the signature that has an **uns32** argument is no longer visible.

Note that signatures do not have to be unique. Consider the following declarations:

```
procedure proc1( i:int32 ); external;
procedure proc2( i:int32 ); external;

proc
   prc : overloads proc1;
   prc : overloads proc2;
```

Although *proc1* and *proc2* have the same argument list (that is, the number of parameters and types of the parameters match), the **overloads** declarations are legal. This, however, creates an ambiguity: if you invoke "prc(int32Var);" how will HLA differentiate the two calls? The answer is simple: HLA uses the last **overloads** declaration to disambiguate the declarations. This might seem confusing and a poor design decision, but this was a conscious design decision. To understand why HLA does this, consider the following example:

```
program example;
   static
      i:int32;

   proc
      proc1 :procedure( i:int32 ); external;
      proc2 :procedure ( i:int32); external;

      prc : overloads proc1;

   procedure local;
   proc
      prc :overloads proc2;
   begin local;

      prc( i ); // Calls proc2

   end local;

begin example;

   prc( i ); // calls proc1

end example;
```

Within a function, a local declaration of an **overloads** identifier can hide a global invocation that has the same signature. As the above example demonstrates, the global declaration is visible again beyond the body of the local function.

When you invoke an **overloads** function, HLA applies a multi-step signature-matching algorithm to determine which of the overloaded functions to call. First, the number of parameters must exactly agree with some **overloads** declaration or HLA will reject the invocation.

HLA compares the invocation parameter list against the list of **overloads** declaration. It scans the list backwards from the last **overloads** declaration (for a given **overloads** identifier) through to the first identifier. If an exact match to the parameter's types is found, then HLA will call that **procedure, method,** or **iterator**.

If HLA scans through the complete list of overloaded function names and doesn't come up with an exact signature match, then it will make a second pass through the list and relax the parameter matching requirements to see if a match is possible. The relaxation takes the following form:

Smaller unsXX types are "promoted" to larger types (e.g., an **uns8** actual parameter can be passed to a function with an **uns32** formal parameter).

Hexadecimal types (byte, word, dword, qword, tbyte, and lword) are compatible with all integer and unsigned types of the same size.

Note that the x86's registers are hexadecimal types.

Dwords are compatible with pointers.

Pointer constants are compatible with strings.

Because relaxing the type checking can produce ambiguity, the "last overloads to first overloads" disambiguation rule apples. Therefore, you should declare higher priority **overloads** declarations last in the **proc** section.

The overloads declaration takes two forms:

```
ovldName : overloads functionName;
ovldName : overloads functionName ( "string calling sequence" );
```

When you use the first form (as in all the examples to this point), invoking the overloaded name (*ovldName*) substitutes the actual function name (*functionName*) whose parameter list matches the actual call. That is, given a declaration of the form:

```
prc : overloads proc1;  // Assume: procedure proc1( u:uns32 );
```

And given the invocation:

```
prc( uns32Var );
```

HLA will generate the following actual procedure call:

```
proc1( uns32Var );
```

In certain circumstances, calling the function using the declared function name is insufficient. For example, if you have a class variable (an object), then you'll actually need to invoke the procedure using the actual object name, not by simply invoking the class procedure, method, or iterator name. The second form of the overloads declaration allows you to specify a string that HLA will expand when calling the actual function. Consider the following declarations inside a namespace:

```
namespace ns;

    procedure a( u:uns32 );
    begin u;
        //…
    end u;

    procedure b( i:int32 );
```

```
    begin b;
        //…
    end b;

    proc
        ab:overloads a;
        ab:overloads b;

end ns;
```

This set of overloads declarations will not work outside the namespace. You would like to be able to invoke *ab* using "ns.ab(uns32Var);" or "ns.ab(int32Var);" but this will not work outside the namespace because an invocation of the form "ns.ab(uns32Var);" produces a call of the form "a(uns32Var);". Of course, outside the namespace this will either generate a syntax error (symbol not found or mismatched parameter list) or it will call the wrong procedure (a procedure named *a* outside the namespace). The solution is to use the second declaration form and explicitly specify the namespace name, as follows:

```
namespace ns;

    procedure a( u:uns32 );
    begin u;
        //…
    end u;

    procedure b( i:int32 );
    begin b;
        //…
    end b;

    proc
        ab:overloads a ( "@global:ns.a" );
        ab:overloads b ( "@global:ns.b" );

end ns;
```

The "@global:" prefix exists in case you invoke the *ns.ab* overloaded procedure name inside another namespace.

Object references are another matter altogether. Consider the following declarations:

```
type
    class c;

        procedure a( u:uns32 );
        begin u;
            //…
        end u;

        procedure b( i:int32 );
        begin b;
            //…
        end b;

        proc
            ab:overloads a;
            ab:overloads b;
```

```
    endclass;

static
    cv :c;
```

An invocation of the form "cv.ab(uns32Var);" produces the call "a(uns32Var);" which does not call the class procedure. Unfortunately, supplying a string operand of the form "c.a" in the **overloads** declaration will not solve the problem. Because a class procedure can be called using several different object variables (and, in the case of class procedures, using the class name), you have to supply the actual object name as part of the invocation string. Fortunately, HLA supplies a compile-time function, **@curVar**, which returns a string containing the current full variable name (including the overloaded name). The following declaration shows how to use **@curVar** to achieve this:

```
type
    c:class

        procedure a(u:uns32); external;
        procedure b(i:int32); external;

        proc
            ab :overloads a
               (
                  "@text( @substr( @curvar, 0, @length( @curvar )-2)" +
                     """a"" )"
               );
            ab :overloads b
               (
                  "@text( @substr( @curvar, 0, @length( @curvar )-2)" +
                     """b"" )"
               );

    endclass;
```

With this class declaration, an invocation of the form

```
cv.ab(uns32Var);
```

produces the call

```
@text(@substr( @curvar, 0, @length( @curvar )-2)+"a")(uns32Var);
```

which expands to

```
cv.a(uns32Var);
```

Note that `@curvar` in this example returns the string "cv.ab". The `@substring` function strips the last two characters from the string (the name of the overloaded function) and then appends "a" or "b" to the string to produce a call to the appropriate function. Because this string is a bit unwieldy, the HLA Standard Library's *hla.hhf* module includes a macro named "ovrldStr" that you can use to generate this text for you. This macro requires a single argument that is the name of the function you want to overload (`a` or `b` in this example). Here's what the above code would look like when using this macro:

```
type
```

```
c:class

   procedure a(u:uns32); external;
   procedure b(i:int32); external;

   proc
      ab :overloads a( hla.ovrldStr( a ));
      ab :overloads b( hla.ovrldStr( b ));

endclass;
```

Of course, you must #include the *hla.hhf* header file in order to use this macro.

For those who are interested in how things work internally, the `hla.ovrldStr` macro takes the following form:

```
#macro ovrldStr( string theFunc );

   "@text( @left( @curvar, @rindex( @curvar, 0, "".""")+1) +
                              """ + theFunc +""" )"

#endmacro
```

Because of the limitations of HLA's implementation languages (Flex and Bison), there are some limitations to how well HLA can recognize various parameter signatures. These problems will be corrected in HLA v3.0. In the meantime, if you run into any problems, you can easily work around the issue(s) by directly calling the procedure or by explicitly type-casting the actual arguments.

11.3 The _vars_ and _parms_ Constants and the _display_ Array

To help those who insist on constructing the activation record themselves, HLA declares two local constants within each procedure: `_vars_` and `_parms_`. The `_vars_` symbol is an integer constant that specifies the number of bytes of local variables declared in the procedure. This constant is useful when allocating storage for your local variables. The `_parms_` constant specifies the number of bytes of parameters. You would normally supply this constant as the parameter to a **ret**() instruction to automatically clean up the procedure's parameters when it returns.

Example:

```
procedure demoVarsParms( parm1:int32; parm2:dword ); @nodisplay; @noframe;
var
   var1:dword;
   var2:dword;
begin demoVarsParms;

   push( ebp );
   mov( esp, ebp );
   sub( _vars_, esp );
      .
      .
      .
   mov( ebp, esp );
   pop( ebp );
   ret( _parms_ );

end demoVarsParms;
```

If you do not specify **@**`nodisplay`, then HLA defines a run-time variable named `_display_` that is an array of pointers to activation records. For more details on the `_display_` variable, see the section on lexical scope.

11.4 External Procedure Declarations

You can declare external procedures (procedures defined in other HLA units or written in languages other than HLA) using the following syntaxes:

```
Original Syntax:

procedure externProc1 (optional parameters) ; options; external;

procedure externProc2 (optional parameters) ; options; external(
"external_name" );


New Style Syntax:

proc
   externProc1 :procedure(optional parameters) {options}; external;
   externProc2 :procedure(optional parameters) {options}; external(
"external_name" );
endproc;
```

As with normal procedure declarations, the parameter list and the procedure options are optional. Note that **@external** and **external** are synonyms. **@external** is deprecated so you should use **external** in new code.

The first form is generally used for HLA-written functions. HLA will use the procedure's name (*externProc1* in this case) as external name.

The second form lets you refer to the procedure by one name within your HLA program (`externProc2` in this case) and by a different name ("*external_name*" in this example) in the generated object code. This second form has two main uses: (1) if you choose an external procedure name that just happens to conflict with a back-end assembler reserved word, the program may compile correctly but fail to assemble. Changing the external name to something else solves this problem. (2) When calling procedures written in external languages you may need to specify characters that are not legal in HLA identifiers. For example, Win32 API calls often use names like *WriteFile@24* containing illegal (in HLA) identifier symbols. The string operand to the external option lets you specify any name you choose. Of course, it is your responsibility to see to it that you use identifiers that are compatible with the external environment; HLA doesn't check these names.

External procedure declarations do not allow the same set of procedure options as regular procedure declarations. In particular, only those options that affect the calling sequence (rather than affecting the code generation within the procedure itself) are legal in an external declaration. The procedure options that are legal within an external procedure declaration are:

```
@use Reg32
@returns( "string" )
@cdecl
@pascal
@stdcall
```

If you declare an external procedure and then declare that same procedure in the same source file, that procedure name becomes *public* and is accessible to source files outside the current file. When declaring the public procedure body (after the appearance of the external procedure

declaration prototype earlier in the source file), the legal procedure options are those that affect the generation of code within the procedure; those that specify how HLA generates code to call the procedure are illegal (that is, the set of procedure options between the external declaration and the procedure's actual declaration are mutually exclusive. If an external procedure declaration appears in a source file, then the actual procedure declaration's procedure options are limited to the following:

```
@noframe, @frame
@nodisplay, @display
@noalignstack, @alignstack
@align( int_constant )
@leave, @noleave
@enter, @noenter
```

11.5 Forward Procedure Declarations

A forward procedure declaration provides a way to declare a procedure *prototype* that allows you to specify the calling syntax for a procedure before actually declaring the body of that procedure. In HLA, the syntax for a forward declaration (procedure prototype) is the following:

```
Original Syntax:

procedure forwardProc (optional parameters) ; options; forward;


New Style Syntax:

proc
    forwardProc :procedure(optional parameters) {options}; forward;
```

The `forward` declaration syntax is necessary because HLA requires all procedure symbols to be declared before they are used. In a few rare cases (where mutual recursion occurs between two or more procedures), it may be impossible to write your code such that every procedure is declared before the first call to the code. More commonly, sorting your procedures to ensure that all procedures are written before their first call may force an artificial organization on the source file, making it harder to read. The forward procedure declaration handles this situation for you. It lets you create a procedure prototype that describes how the procedure is to be called without actually specifying the procedure body. Later on in the source file, the full procedure declaration must appear.

Note: an external declaration also serves as a forward declaration. If you have an external definition at the beginning of your program (perhaps it appears in an include file), you do not need to provide a forward declaration as well.

As for external procedure declarations, forward declarations do not allow the same set of procedure options as regular procedure declarations. In particular, only those options that affect the calling sequence (rather than affecting the code generation within the procedure itself) are legal in a forward declaration. The procedure options that are legal within a forward procedure declaration are:

```
@use Reg32
@returns( "string" )
@cdecl
@pascal
@stdcall
```

When the procedure declaration appears later in the source file, the legal procedure options are

```
@noframe, @frame
@nodisplay, @display
@noalignstack, @alignstack
@align( int_constant )
@leave, @noleave
@enter, @noenter
```

Note that the presence of a forward declaration in a source file does not make the procedure name public. Also, note that you may not have both a forward and external procedure declaration for the same procedure name.

11.6 Setting Default Procedure Options

By default, HLA does the following:

- Creates a display for every procedure
- Emits code to construct the stack frame for each procedure
- Emits code to align ESP on a four-byte boundary upon procedure entry
- Assumes that it cannot modify any register values when passing (non-register) parameters
- The first instruction of the procedure is unaligned.

These options are the most general and "safest" for beginning assembly language programmers. However, the code HLA generates for this general case may not be as compact or as fast as is possible in a specific case. For example, few procedures will actually need a display data structure built upon procedure activation. Therefore, the code that HLA emits to build the display can reduce the efficiency of the program. Advanced programmers, of course, can use procedure options like **@nodisplay** to tell HLA to skip the generation of this code. However, if a program contains many procedures and none of them requires a display, continually adding the **@nodisplay** option can get annoying. Therefore, HLA allows you to treat these directives as "pseudo-compile-time-variables" to control the default code generation. E.g.,

```
?@display := true; // Turns on default display generation.
?@display := false; // Turns off default display generation.
?@nodisplay := true; // Turns off default display generation.
?@nodisplay := false; // Turns on default display generation.

?@frame := true; // Turns on default frame generation.
?@frame := false; // Turns off default frame generation.
?@noframe := true; // Turns off default frame generation.
?@noframe := false; // Turns on default frame generation.

?@alignstack := true; // Turns on default stk alignment code generation.
?@alignstack := false; // Turns off default stk alignment code generation.
?@noalignstack := true; // Turns off default stk alignment generation.
?@noalignstack := false; // Turns on default stk alignment generation.

?@enter := true; // Turns on default ENTER code generation.
?@enter := false; // Turns off default ENTER code generation.
?@noenter := true; // Turns off default ENTER code generation.
?@noenter := false; // Turns on default ENTER code generation.

?@leave := true; // Turns on default LEAVE code generation.
?@leave := false; // Turns off default LEAVE code generation.
?@noleave := true; // Turns off default LEAVE code generation.
?@noleave := false; // Turns on default LEAVE code generation.
```

```
?@align := 1; // Turns off procedure alignment (align on byte boundary).
?@align := int_expr; // Sets alignment, must be a power of two.
```

These directives may appear anywhere in the source file. They set the internal HLA default values and all procedure declarations following one of these assignments (up to the next, corresponding assignment) use the specified code generation option(s). Note that you can override these defaults by using the corresponding procedure options mentioned earlier.

11.7 Disabling HLA's Automatic Code Generation for Procedures

Before jumping in and describing how to use the high-level HLA features for procedures, the best place to start is with a discussion of how to disable these features and write "plain old fashioned" assembly language code. This discussion is important because procedures are the one place where HLA automatically generates a lot of code for you and many assembly language programmers prefer to control their own destinies; they don't want the compiler to generate any excess code for them. So disabling HLA's automatic code generation capabilities is a good place to start.

By default, HLA automatically emits code at the beginning of each procedure to do five things:

(1) Preserve the pointer to the previous activation record (EBP);

(2) build a display in the current activation record;

(3) allocate storage for local variables;

(4) load EBP with the base address of the current activation record;

(5) adjust the stack pointer (downwards) so that it points at a double-word-aligned address.

When you return from a procedure, by default HLA will deallocate the local storage and return, removing any parameters from the stack.

To understand the code that HLA emits, consider the following simple procedure:

```
procedure p( j:int32 );
var
    i:int32;
begin p;
end p;
```

Here is a dump of the symbol table that HLA creates for procedure p:

```
p               <0,proc>:Procedure type (ID=p__hla_1) parms:4
    --------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =4
    i               <1,var >:int32, (4 bytes, ofs:-12)
    _parms_         <1,cons>:uns32, (4 bytes)  =4
    _display_       <1,var >:dword, (8 bytes, ofs:-4)
    j               <1,valp>:int32, (4 bytes, ofs:8)
    _finalize_      <1,val >:string, (0 bytes)  =""
    _initialize_    <1,val >:string, (0 bytes)  =""
    p               <1,proc>:
-----------------------------------
```

The important thing to note here is that local variable `i` is at offset -12 and HLA automatically created an 8-byte local variable named `_display_` which is at offset -4 (note: the *_display_* variable uses negative indexes, so the two 4-byte elements are at offset -4 [index 0] and -8 [index -1]).

HLA, with the "-hla" and "-source" command-line parameters, produces the following pseudo-HLA code for the procedure above (annotations in italics are not emitted by HLA, this output is subject to changes in HLA code generation algorithms; the output is pure assembly language with no "hidden" or high-level code):

```
procedure p__hla_2;
begin p__hla_2;
        push( ebp );;Dynamic link (pointer to previous activation record)
        push( [ebp-4] );;Display for lex level 0
        lea( [esp+4], ebp );;Get frame ptr (point EBP at current
activation record)
        push( ebp );;Ptr to this proc's A.R. (part of display
construction)
        sub( 4, esp );;Local storage.
        and( -4, esp );;dword-align stack

xp__hla_2__hla_:
        mov( ebp, esp );;Deallocate local variables.
        pop( ebp );;Restore pointer to previous activation record.
        ret( 4 );;Return, popping parameters from the stack.
end p__hla_2;
```

Building the display data structure is not very common in standard assembly language programs. This is only necessary if you are using nested procedures and those nested procedures need to access non-local variables. Since this is a rare situation, many programmers will immediately want to tell HLA to stop emitting the code to generate the display. This is easily accomplished by adding the @nodisplay procedure option to the procedure declaration. Adding this option to procedure *p* produces the following:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
end p;
```

Compiling this procedures the following symbol table dump:

```
Symbol Table:

p               <0,proc>:Procedure type (ID=p__hla_1) parms:4
    --------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =4
    i               <1,var >:int32, (4 bytes, ofs:-4)
    _parms_         <1,cons>:uns32, (4 bytes)  =4
    j               <1,valp>:int32, (4 bytes, ofs:8)
    _finalize_      <1,val >:string, (0 bytes)  =""
    _initialize_    <1,val >:string, (0 bytes)  =""
    p               <1,proc>:
-----------------------------------
```

Note that the _display_ variable is gone and the local variable i is now at offset -4. Here is the code that HLA emits for this new version of the procedure:

```
procedure p__hla_2;
begin p__hla_2;
        push( ebp );;Save ptr to previous activation record.
        mov( esp, ebp );;Point EBP at current activation record.
        sub( 4, esp );;Local storage.
        and( -4, esp );;Align stack on dword boundary.
```

```
; Exit point for the procedure:

xp__hla_2__hla_:
        mov( ebp, esp );;Deallocate local variables.
        pop( ebp );;Restore pointer to previous activation record.
        ret( 4 );;Return, popping parameters from the stack.
end p__hla_2;
```

As you can see, this code is smaller and a bit less complex. Unlike the code that built the display, it is common for an assembly language programmer to construct an activation record in a manner similar to this. Indeed, about the only instruction out of the ordinary above is the "AND" instruction that double-word-aligns the stack (OS calls require the stack to be double-word-aligned, and the system performance is much better if the stack is double-word aligned).

This code is still relatively inefficient if you don't pass parameters on the stack and you don't use automatic (non-static, local) variables. Many assembly language programmers pass their few parameters in machine registers and maintain local values in the registers. If this is the case, then the code above is pure overhead. You can inform HLA that you wish to take full responsibility for the entry and exit code by using the @noframe procedure option. Consider the following version of p:

```
procedure p( j:int32 ); @nodisplay; @noframe;
var
    i:int32;
begin p;
end p;
```

(This produces the same symbol table dump as the previous example).

HLA emits the following code for this version of p:

```
procedure p__hla_2;
begin p__hla_2;
end p__hla_2;
```

Whoa! There's nothing there! But this is exactly what the advanced assembly language programmer wants. With both the @nodisplay and @noframe options, HLA does not emit any extra code for you. You would have to write this code yourself.

By the way, you *can* specify the @noframe option without specifying the @nodisplay option. HLA still generates no extra code, but it will assume that you are allocating storage for the display in the code you write. That is, there will be an 8-byte _display_ variable created and i will have an offset of -12 in the activation record. It will be your responsibility to deal with this. Although this situation is possible, it's doubtful this combination will be used much at all.

Note a major difference between the two versions of p when @noframe is not specified and @noframe is specified: if @noframe is not present, HLA automatically emits code to return from the procedure. This code executes if control falls through to the "end p;" statement at the end of the procedure. Therefore, if you specify the @noframe option, you must ensure that the last statement in the procedure is a **ret**() instruction or some other instruction that causes an unconditional transfer of control. If you do not do this, then control will fall through to the beginning of the next procedure in memory, probably with unintended results.

The **ret**() instruction presents a special problem. It is dangerous to use this instruction to return from a procedure that does not have the @noframe option. Remember, HLA has emitted code that pushes much data onto the stack. If you return from such a procedure without first removing this data from the stack, your program will probably crash. The correct way to return from a procedure without the @noframe option is to jump to the bottom of the procedure and run off the end of it. Rather than require you to explicitly put a label into your program and jump to this label, HLA provides the **exit** *procname*; instruction. HLA compiles the **exit** instruction into a **jmp** that

transfers control to the clean-up code HLA emits at the bottom of the procedure. Consider the following modification of p and the resulting assembly code produced:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
    exit p;
    nop();
end p;


procedure p__hla_2;
begin p__hla_2;

        push( ebp );
        mov( esp, ebp );
        sub( 4, esp );
        and( -4, esp );
        jmp xp__hla_2__hla_;
        nop;
xp__hla_2__hla_:
        mov( ebp, esp );
        pop( ebp );
        ret( 4 );
end p__hla_2;
```

As you can see, HLA automatically emits a label to the assembly output file (*xp__hla_2__hla_* in this instance) at the bottom of the procedure where the clean-up code starts. HLA translates the "**exit** p;" instruction into a **jmp** to this label.

If you look back at the code emitted for the version of p with the **@**noframe option, you'll note that HLA did not emit a label at the bottom of the procedure. Therefore, HLA cannot generate a jump to this nonexistent label, so you cannot use the exit statement in a procedure with the **@**noframe option (HLA will generate an error if you attempt this).

Of course, HLA will *not* stop you from putting a **ret**() instruction into a procedure without the **@**noframe option (some people who know exactly what they are doing might actually want to do this). Keep in mind, if you decide to do this, that you must deallocate the local variables (that's what the "mov esp, ebp" instruction is doing), you need to restore EBP (via the "pop ebp" instruction above), and you need to deallocate any parameters pushed on the stack (the "ret 4" handles this in the example above). The following code demonstrates this:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;

    if( j = 0 ) then

        // Deallocate locals.

        mov( ebp, esp );

        // Restore old EBP

        pop( ebp );
```

```
        // Return and pop parameters

        ret( 4 );

    endif;
    nop();
end p;


procedure p__hla_2;
begin p__hla_2;

        push( ebp );
        mov( esp, ebp );
        sub( 4, esp );
        and( -4, esp );
        cmp( 0, (type dword [ebp+8]) );
        jne false__hla_3;
        mov( ebp, esp );
         pop( ebp );
        ret( 4 );
false__hla_3:
        nop;
xp__hla_2__hla_:
        mov( ebp, esp );
        pop( ebp );
        ret( 4 );
end p__hla_2;
```

If "real" assembly language programmers would generally specify both the @noframe and @nodisplay options, why not make them the default case (and use **@frame** and **@display** options to specify the generation of the activation record and display)? Well, keep in mind that HLA was originally designed as a tool to teach assembly language programming to beginning students. Those students have considerable difficulty comprehending concepts like activation records and displays. Having HLA generate the stack frame code and display generation code automatically saves the instructor from having to teach (and explain) this code. Even if the student never uses a display, it doesn't make the program incorrect to go ahead and generate it. The only real cost is a little extra memory and a little extra execution time. This is not a problem for beginning students who haven't yet learned to write efficient code. Therefore, HLA was optimized for the beginner at the expense of the advanced programmer. It is also worthwhile to point out that the behavior of the EXIT statement depends upon displays if you attempt to exit from a nested procedure; yet another reason for HLA's default behavior. Of course, you can always override HLA's default behavior by using the @nodisplay and @noframe compile-time variablesw.

If you are absolutely certain that your stack pointer is aligned on a four-byte boundary upon entry into a procedure, you can tell HLA to skip emitting the **and**($FFFF_FFFC, ESP); instruction by specifying the @noalignstack procedure option. Note that specifying @noframe also specifies @noalignstack.

11.8 Procedure Calls and Parameters in HLA

HLA's high-level support consists of three main features: HLL-like declarations, the HLL statements (IF, WHILE, etc), and HLA's support for procedure calls and parameter passing. This section discusses the syntax for procedure declarations and how HLA generates code to automatically pass parameters to a procedure.

The syntax for HLA procedure declarations was touched on earlier; however, it's probably a good idea to review the syntax as well as describe some options that previous sections ignored. There are several procedure declaration forms; the following examples demonstrate them all[1]:

```
// Standard procedure declaration:

procedure procname (opt_parms); proc_options
begin procname;
    << procedure body >>
end procname;

// New style procedure declaration:

proc
    procname :procedure(opt_parms); proc_options
    begin procname;
        << procedure body >>
end procname;

// External procedure declarations:

procedure extname (opt_parms); proc_options external;
procedure extname (opt_parms); proc_options external( "name");

// New style external procedure declarations:

proc
    extname :procedure(opt_parms) {proc_options}; external;
    extname :procedure(opt_parms) {proc_options}; external( "name");
endproc;

// Original forward procedure declarations:

procedure fwdname (opt_parms); proc_options forward;

// New style forward procedure declarations:

proc
    fwdname :procedure(opt_parms) {proc_options}; forward;
```

`Opt_parms` indicates that the parameter list is optional; the parentheses are not present if there are no parameters present.

Proc_options is any combination (zero or more) of the procedure options (see the discussion earlier for these options)

11.9 Calling HLA Procedures

There are two standard ways to call an HLA procedure: use the **call** instruction or simply specify the name of the procedure as an HLA statement. Both mechanisms have their plusses and minuses.

1. This section only discusses procedure declarations. Other sections will describe iterators and methods.

To call an HLA procedure using the **call** instruction is exceedingly easy. Simply use either of the following syntaxes:

```
call( procName );
call procName;
```

Either form compiles into an 80x86 **call** instruction that calls the specified procedure. The difference between the two is that the first form (with the parentheses) returns the procedure's "returns" value, so this form can appear as an operand to another instruction. The second form above always returns the empty string, so it is not suitable as an operand of another instruction. Also, note that the second form requires a statement or procedure label, you may not use memory-addressing modes in this form; on the other hand, the second form is the only form that lets you "call" a statement label (as opposed to a procedure label); this form is useful on occasion.

If you use the **call** statement to call a procedure, then you are responsible for passing any parameters to that procedure. In particular, if the parameters are passed on the stack, you are responsible for pushing those parameters (in the correct order) onto the stack before the call. This is a lot more work than letting HLA push the parameters for you, but in certain cases you can write more efficient code by pushing the parameters yourself.

The second way to call an HLA procedure is to simply specify the procedure name and a list of actual parameters (if needed) for the call. This method has the advantage of being easy and convenient at the expense of a possible slight loss in efficiency and flexibility. This calling method should also prove familiar to most HLL programmers. As an example, consider the following HLA program:

```
program parameterDemo;

#include( "stdio.hhf" );

procedure PrtAplusB( a:int32; b:int32 ); @nodisplay;
begin PrtAplusB;

    mov( a, eax );
    add( b, eax );
    stdout.put( "a+b=", (type int32 eax ), nl );

end PrtAplusB;

static
    v1:int32 := 25;
    v2:int32 := 5;

begin parameterDemo;

    PrtAplusB( 1, 2 );
    PrtAplusB( -7, 12 );
    PrtAplusB( v1, v2 );

    mov( -77, eax );
    mov( 55, ebx );
    PrtAplusB( eax, ebx );

end parameterDemo;

This program produces the following output:

a+b=3
a+b=5
a+b=30
```

```
a+b=-22
```

As you can see, call `PrtAplusB` in HLA is very similar to calling procedures (and passing parameters) in a high-level language like C/C++ or Pascal. There are, however, some key differences between and HLA call and a HLL procedure call. The next section will cover those differences in detail. The important thing to note here is that if you choose to call a procedure using the HLL syntax (that is, the second method above), you will have to pass the parameters in the parameter list and let HLA push the parameters for you. If you want to take complete control over the parameter passing code, you should use the **call** instruction.

11.10 Parameter Passing in HLA, Value Parameters

The previous section probably gave you the impression that passing parameters to a procedure in HLA is nearly identical to passing those same parameters to a procedure in a high-level language. The truth is, the examples in the previous section were rigged. There are actually many restrictions on how you can pass parameters to an HLA procedure. This section discusses the parameter passing mechanism in detail.

The most important restriction on actual parameters in a call to an HLA procedure is that HLA only allows memory variables, registers, constants, and certain other special items as parameters. In particular, you cannot specify an arithmetic expression that requires computation at run-time (although a constant expression, computable at compile time is okay). The bottom line is this: if you need to pass the value of an expression to a procedure, you must compute that value prior to calling the procedure and pass the result of the computation; HLA will not automatically generate the code to compute that expression for you.

The second point to mention here is that HLA is a strongly typed language when it comes to passing parameters. This means that with only a few exceptions, the type of the actual parameter must exactly match the type of the formal parameter. If the actual parameter is an `int8` object, the formal parameter had better not be an `int32` object or HLA will generate an error. The only exceptions to this rule are the **byte, word, dword, qword, tbyte**, and **lword** types. If a formal parameter is of type **byte**, the corresponding actual parameter may be any one-byte data object. If a formal parameter is a **word** object, the corresponding actual parameter can be any two-byte object. Likewise, if a formal parameter is a **dword** object, the actual parameter can be any four-byte data type. And so on… Conversely, if the actual parameter is a **byte, word, dword, qword, tbyte,** or **lword** object, it can be passed without error to any one, two, four, eight, ten, or sixteen-byte actual parameter (respectively). Programmers who are lazy make all their parameters one of these hexadecimal types (at least, wherever possible). Programmers who care about the quality of their code use untyped parameters cautiously.

For efficiency reasons (dictated by the operating system and the Intel ABI), HLA procedure calls always pass all parameters as a multiple of four bytes. When passing a byte-sized parameter on the stack by value, the actual parameter value consumes the L.O. byte of the double word passed on the stack. The function ignores the H.O. three bytes of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. three bytes if it is not inconvenient to do so.

11.10.1 Passing Byte-Sized Parameters by Value

When passing a byte-sized constant, you should simply push a double-word containing the 8-bit value, e.g,

```
pushd( 5 );
call someSubroutine;
```

When passing the 8-bit value of the 8-bit registers AL, BL, CL or DL onto the stack, you should simply push the 32-bit register that holds the 8-bit register, e.g.,

```
push( eax );  // Pushes AL onto the stack
call someSubroutine;
push( ebx );  // Pushes BL onto the stack
call someOtherSubroutine;
```

Note that this trick does not apply to the AH, BH, CH, or DH registers. The best code to use when you need to push these registers is to drop the stack down by four bytes and then move the desired register into the memory location you've just created on the stack, e.g.,

```
sub( 4, esp );
mov( AH, [esp] ); // Pushes AH onto the stack
call someSubroutine;
sub( 4, esp );
mov( BH, [esp] ); // Pushes BH onto the stack
call someOtherSubroutine;
```

Here's another way you can accomplish this (a little slower, but leaves zeros in the H.O. three bytes):

```
pushd( 0 );
mov( CH, [esp] ); // Pushes CH onto the stack
call someSubroutine;
```

When passing a byte-sized variable, you should try to push the variable's value and the following three bytes, using code like the following:

```
pushd( (type dword eightBitVar ));
call someSubroutine;
```

There is one drawback to the approach above. In three very rare cases the code above could cause a segmentation fault. If the 8-bit variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, if a register is available, is to move the 8-bit value into AL, BL, CL, or DL and push the corresponding 32-bit register. If no registers are available, then you can write code like the following:

```
push( eax );
push( eax );
mov( byteVar, al );
mov( al, [esp+4] );
pop( eax );
call someSubroutine;
```

This code is ugly and slightly inefficient, but it will always work (assuming, of course, you don't get a stack overflow).

The HLA compiler will generate code similar to this last example if you pass a static byte variable as the actual parameter to a library function expecting an 8-bit value parameter:

```
someLibraryRoutine( byteVar );
```

Therefore, if efficiency is a concern to you, you should try to load the byte variable (byteVar in this example) into AL, BL, CL, or DL prior to calling someLibraryRoutine, e.g.,

```
mov( boolVar, al );
someLibraryRoutine( al );
```

Note that HLA will push a whole double-word if the actual parameter is an automatic variable or some other parameter (both of which are allocated on the stack). In this case, you're generally guaranteed that the three bytes following the byte variable in memory are in readable memory space. It is possible to set up the EBP register to violate this assumption, but HLA assumes that you're not trying to cause a segmentation fault and assumes that it's safe to access those extra three bytes beyond the byte variable. Here is an example of the code generation for various byte parameters:

```
program t;
static
```

```
    b:byte;

procedure p( b0:byte ); @nodisplay;

    procedure q( b1:byte; b2:byte; b3:byte; b4:byte; b5:byte; b6:byte;
b7:byte );
    begin q;
    end q;

var
    b8  :byte;
    w   :word;
    b9  :byte;

begin p;

    q( b, b0, b8, b9, al, ah, 255 );

end p;

begin t;
end t;
```

Here's the code generation for the call to the *q* procedure (MASM syntax, as output by HLA v2.2). Note that the comments in italics are not emitted by HLA:

```
 ; Push static variable b onto the stack

pushd( 0 );
push( eax );
mov( b__hla_1, al );
mov( al, [esp+4] );
pop( eax );

 ; Push parameter b0 onto the stack:

push( (type dword [ebp+8]) );

 ; Push automatic variable b8 onto the stack:

pushd( 0 );
push( eax );
mov( [ebp-1], al );
mov( al, [esp+4] );
pop( eax );

 ; Push automatic variable b9 onto the stack:

push( (type dword [ebp-4]) );

 ; Push AL onto the stack:

push( eax );

 ; Push AH onto the stack:
```

```
sub( 4, esp );
mov( ah, [esp] );

 ; Push 255 onto the stack:

pushd( 255 );

 ; Call q

call q__hla_3;
```

Note the difference in code generation between the *b8* and *b9* local variables. Because *b8* has an offset of -1 in the activation record and HLA is playing it safe and only accessing a single byte at [EBP-1]. This prevents any access to bytes that have a positive or zero index off of EBP. It generates slightly better code if the variable's index is -4 or less because it can safely push four bytes and all four bytes will have a negative offset from EBP. Moral of the story: if you intend to use the HLA HLL-like calling sequence and you intend to pass local (automatic) variables as byte parameters, try to ensure that those byte variables have an offset of -4 or less in the activation record.

If one of the EAX, EBX, ECX, or EDX registers is always free and available when calling a procedure with byte-sized parameters, you can often improve the quality of the code that HLA generates by attaching an **@uses** procedure option to the procedure's declaration. Consider the following modification to the above code:

```
program t;
static
   b:byte;

procedure p( b0:byte ); @nodisplay;

   procedure q
   (
      b1 :byte;
      b2 :byte;
      b3 :byte;
      b4 :byte;
      b5 :byte;
      b6 :byte;
      b7 :byte
   ); @use ecx;
   begin q;
   end q;

var
   b8 :byte;
   w  :word;
   b9 :byte;

begin p;

   q( b, b0, b8, b9, al, ah, 255 );

end p;

begin t;
```

```
end t;
```

The **@use ecx;** clause tells HLA that it can use the ECX register, wiping out its contents, if HLA finds it convenient to do so. Compare the code HLA generates for the call to *q* above against the earlier versions:

```
; Push static variable b onto the stack

mov( b__hla_1, cl );
push( ecx );


 ; Push parameter b0 onto the stack:

push( (type dword [ebp+8]) );

 ; Push automatic variable b8 onto the stack:

mov( [ebp-1], cl );
push( ecx );

 ; Push automatic variable b9 onto the stack:

push( (type dword [ebp-4]) );

 ; Push AL onto the stack:

push( eax );

 ; Push AH onto the stack:

sub( 4, esp );
mov( ah, [esp] );

 ; Push 255 onto the stack:

pushd( 255 );

 ; Call q

call q__hla_3;
```

If you want to use the high-level calling sequence, but you don't like the inefficient code HLA sometimes produces when generating code to pass your byte-sized parameters, you can always use the #{...}# sequence parameter to override HLA's code generation and substitute your own code for one or two parameters. Of course, it doesn't make any sense to pass all the parameters in a procedure using this trick, it would be far easier just to use the call instruction. Example:

```
q( b, b0, #{push( (type dword b8) );}#, b9, al, ah, 255 );
```

If efficiency is a concern to you and the **@use reg$_{32}$** procedure option isn't acceptable (perhaps because you can't guarantee that a register is always available), you should try to load a byte variable you want to pass as a parameter into AL, BL, CL, or DL prior to calling the subroutine, e.g.,

```
mov( byteVar, al );
```

```
someSubroutine( al );
```

11.10.2 Passing Word-Sized Parameters by Value

When passing a word-sized parameter on the stack by value, the actual parameter value consumes the L.O. two bytes of the double word passed on the stack. The function ignores the H.O. word of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. word if it is not inconvenient to do so.

When passing a word-sized constant, you should simply push the double word containing the 16-bit value, e.g.,

```
pushd( 5 );
call someSubroutine;
```

When passing the 16-bit value of a 16-bit register (AX, BX, CX, DX, SI, DI, BP, or SP) onto the stack, you should simply push the 32-bit register that holds the 16-bit register, e.g.,

```
push( eax );  // Pushes AX onto the stack
call someSubroutine;
push( ebx );  // Pushes BX onto the stack
call someOtherSubroutine;
```

When passing a word-sized variable, you should try to push the variable's value and the following two bytes, using code like the following:

```
pushd( (type dword sixteenBitVar ));
call someSubroutine;
```

There is one drawback to the approach above. In three very rare cases, the code above could cause a segmentation fault. If the 16-bit variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, is to use two consecutive pushes:

```
pushw( 0 );// H.O. word is zeros
push( sixteenBitVar );
call someSubroutine;
```

The HLA compiler will generate code similar to this last example if you pass a word variable as the actual parameter to a function expecting a 16-bit value parameter:

```
someSubroutine( wordVar );
```

Here is a more complete example:

```
program t;
static
    w:word;

procedure p( w0:word ); @nodisplay;

    procedure q
    (
        w1  :word;
        w2  :word;
```

```
        w3  :word;
        w4  :word;
        w5  :word;
        w6  :word;
        w7  :word
    );
    begin q;
    end q;

var
    w8  :word;
    w   :word;
    w9  :word;

begin p;

    q( w, w0, w8, w9, ax, si, 255 );

end p;

begin t;
end t;
```

This procedures the following (pseudo-HLA) assembly language output:

```
; Push w

pushw( 0 );
push( (type word [ebp-4]) );

; Push w0

push( (type dword [ebp+8]) );

; Push w8

pushw( 0 );
push( (type word [ebp-2]) );

; Push w9

pushw( 0 );
push( (type word [ebp-6]) );

; Push ax

push( eax );

; Push si

push( esi );

; Push 255

pushd( 255 );
```

```
; call q

call q__hla_3;
```

11.10.3 Passing Double-Word-Sized Parameters by Value

Because 32-bit double-word objects are the native x86 data type, there are only a few issues with passing 32-bit parameters on the stack to a standard library routine.

First, and this applies to all stack operations not just 32-bit pushes and pops, you should always keep the stack 32-bit aligned. That is, the value in ESP should always contain a value that is a multiple of four (i.e., the L.O. two bits of ESP must always contain zeros). If this is not the case, many OS API and standard library function calls will fail.

When passing a 32-bit value onto the stack, just about any mechanism you can use to push that value is perfectly valid. You can efficiently push constants, registers, and memory locations using a single push instruction, e.g.,

```
pushd( 12345 ); // Passing a 32-bit constant
push( mem32 );  // Passing a dword variable
push( eax );    // Passing a 32-bit register
call someRoutine;
```

Of course, you can always use the HLA high-level syntax to pass a 32-bit object to a subroutine. HLA automatically generates the appropriate code to pass the dword object as a parameter on the stack. Note that HLA automatically recognizes the lexeme "dx:ax" as a 32-bit value and will push these two registers (DX first, AX second) onto the stack.

11.10.4 Passing Quad-Word-Sized Parameters by Value

Because qword (64-bit) objects are a multiple of 32 bits in size, manually passing qword objects on the stack is very easy. All you need do is push two double-word values. Because the stack grows downward in memory and the x86 is a little endian machine, you must push the H.O. dword first and the L.O. dword second.

If the qword value is held in a register pair, then push the register containing the H.O. dword first and the L.O. dword second. For example, if EDX:EAX contains the 64-bit value, then you'd push the qword as follows:

```
push( edx );  // Push H.O. dword
push( eax );  // Push L.O. dword
call someLibraryRoutine;
```

If the qword value is held in a qword variable, then you must first push the H.O. dword of that variable followed by the L.O. dword, e.g.,

```
push( (type dword qwordVar[4])); // Push H.O. dword first
push( (type dword qwordVar));    // Push L.O. dword second
call someLibraryRoutine;
```

If the qword value you wish to pass is a constant, then you have to compute the L.O. and H.O. dword values for that constant and push those. When using HLA, you can use the compile-time computational capabilities of HLA to do this for you, e.g.,

```
pushd( ((some64bitConst) >> 32);
pushd( ((some64bitConst) & $FFFF_FFFF );
call someLibraryRoutine;
```

If this is something you do frequently, you might want to create a macro to break up the 64-bit value and push it for you.

Of course, you can always use the HLA high-level syntax to pass a 64-bit object to a subroutine. HLA automatically generates the appropriate code to pass the qword object as a parameter on the stack. Note that HLA automatically recognizes the lexeme "edx:eax" as a 64-bit value and will push these two registers (EDX first, EAX second) onto the stack.

11.10.5 Passing Tbyte-Sized Parameters by Value

For efficiency reasons, operating system APIs and HLA standard library routines always pass all parameters as a multiple of four bytes. When passing a **tbyte**-sized parameter on the stack by value, the actual parameter value consumes the L.O. ten bytes of the three double words passed on the stack. The function ignores the H.O. word of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. word if it is not inconvenient to do so.

The following code demonstrates how to pass a ten-byte object to a standard library routine:

```
pushw( 0 ); // Dummy H.O. word of zero
push( (type word tbyteVar[8]));  // Push H.O. byte of tbyte object
push( (type dword tbyteVar[4])); // Push bytes 4-7 of tbyte object
push( (type dword tbyteVar[0])); // Push L.O. dword of tbyte object
call someLibraryRoutine;
```

If your **tbyte** object is not at the very end of allocated memory, you could probably combine the first two instructions in this sequence to produce the following (slightly more efficient) code:

```
push( (type dword tbyteVar[8])); // Pushes two extra bytes.
```

This pushes the two bytes beyond *tbyteVar* onto the stack but, presumably, the function will ignore all bytes beyond the tenth byte passed on the stack, so the actual values in those H.O. two bytes are irrelevant. Note the earlier discussion (in the section on pushing byte parameters) about the rare possibility of a memory access error when using this trick.

Of course, you can always use the HLA high-level syntax to pass an 80-bit object to a standard library routine. HLA automatically generates the appropriate code to pass the **tbyte** object as a 12-byte parameter on the stack.

11.10.6 Passing Lword-Sized Parameters by Value

Because **lword** (128-bit) objects are a multiple of 32 bits in size, manually passing **lword** objects on the stack is very easy. All you need do is push four dword values. Because the stack grows downward in memory and the x86 is a little endian machine, you must push the H.O. dword first and the L.O. dword last.

If the **lword** value is held in an **lword** variable, then you must first push the H.O. dword of that variable followed by the lower-order dwords, down to the L.O. dword, e.g.,

```
push( (type dword qwordVar[12])); // Push H.O. dword first
push( (type dword qwordVar[8]));  // Push bytes 8-11 second
push( (type dword qwordVar[4]));  // Push bytes 4-7 third
push( (type dword qwordVar));     // Push L.O. dword last
call someRoutine;
```

If the **lword** value you wish to pass is a constant, then you have to compute the four dword values for that constant and push those. When using HLA, you can use the compile-time computational capabilities of HLA to do this for you, e.g.,

```
pushd( ((some128bitConst) >> 96);
pushd( ((some128bitConst) >> 64 & $FFFF_FFFF );
pushd( ((some128bitConst) >> 32 & $FFFF_FFFF );
```

```
pushd( ((some128bitConst) & $FFFF_FFFF );
call someRoutine;
```

If this is something you do frequently, you might want to create a macro to break up the 128-bit value and push it for you.

Of course, you can always use the HLA high-level syntax to pass a 128-bit object to a standard library routine. HLA automatically generates the appropriate code to pass the **lword** object as a parameter on the stack.

11.10.7 Passing Large Parameters by Value

When using a HLL-like call, HLA will automatically copy an actual value parameter into local storage for the procedure, regardless of the size of the parameter. If your value parameter is a one-million-byte array, HLA will allocate storage for 1,000,000 bytes and then copy that array in on each call. C/C++ programmers may expect HLA to automatically pass arrays by reference (as C/C++ does) but this is not the case. If you want your parameters passed by reference, you must explicitly state this.

If you're not using a HLL-like call, it is your responsibility to make room for large parameters and copy that parameter data to the stack before a call. Consider the following example that passes a fair-sized array (256 bytes) by value:

```
program t;
type
   array:byte[256];

procedure p( a:array );
begin p;
end p;

static
   theArray:array;

begin t;

   p( theArray );

end t;
```

Here is the code that HLA generates for the call to procedure *p* in the main program above:

```
// Reserve storage for the 256-byte array to be passed on the stack

lea( [esp-256], esp );

// Preserve the registers that rep.movsd uses:

push( esi );
push( edi );
push( ecx );
pushfd;

// Copy the array from the source location (theArray) to
// the storage just allocated on the stack:

cld;
lea( theArray__hla_2, esi );
```

```
mov( 64, ecx );
lea( [esp+16], edi );// Address of array on stack
rep movsd

// Restore the registers:

popfd;
pop( ecx );
pop( edi );
pop( esi );

// Call p
call p__hla_1;
```

The code HLA generates to copy value parameters, while not particularly bad, certainly isn't always optimal. If you need the fastest possible code when passing parameters by value on the stack, it would be better if you explicitly pushed the data yourself.

11.11 Parameter Passing in HLA, Reference, Value/Result, and Result Parameters

The one good thing about pass by reference, pass by value/result, and pass by result parameters is that the parameters are always four byte pointers, regardless of the size of the actual parameter. Therefore, HLA has an easier time generating code for these parameters than it does generating code for pass by value parameters.

In a procedure call HLA treats reference, value/result, and result parameters identically. The code within the procedure is responsible for differentiating these parameter types (value/result and result parameters generally require copying data between local storage and the actual parameter). The following discussion will simply refer to pass by reference parameters, but it applies equally well to pass by value/result and pass by result.

When passing a parameter by reference, you must push the address of the actual parameter (rather than its value) onto the stack. For static objects, you can use the push immediate instruction, e.g., (in HLA syntax):

```
pushd( &staticVar );
call someLibraryRoutine;
```

For automatic variables, or objects whose address is not a simple static offset (e.g., a complex pointer address involving registers and what-not), you'll have to use the LEA instruction to first compute the address and then push that register's value, e.g.,

```
lea( eax, anAutomaticVar );  // Variable allocated on the stack
push( eax );
call someLibraryRoutine;
```

If the variable's address is a simple offset from a single register (such as automatic variables declared in the stack frame and referenced off of the EBP register), you can push the address of the variable by pushing the base register and adding the offset of that variable to the value left on the stack, thusly:

```
push( ebp );  // anAutoVar is found at EPB+@offset(anAutoVar)
add( @offset( anAutoVar ), (type dword [esp]));
call someLibraryRoutine;
```

If the address you want to pass in a reference parameter is a complex address, you'll have to use the LEA instruction to compute that address and push it onto the stack. This, unfortunately, requires a free 32-bit register. If no 32-bit registers are free, you can use code like the following to achieve this:

```
sub( 4, esp );// Reserve space for parameter on stack
push( eax );   // Preserve EAX
lea( eax, [ebp+@offset(autoVar)][ecx*4+3] );
mov( eax, [esp+4] ); // Store in parameter location
pop( eax );    // Restore EAX
call someLibraryRoutine;
```

Of course, it's much nicer to use the HLA high-level syntax for calls like this as the HLA compiler will automatically handle all the messy code generation details for you.

Like high-level languages, HLA places a whopper of a restriction on pass by reference parameters: they can only be memory locations. Constants and registers are not allowed since you cannot compute their address. Do keep in mind, however, that any valid memory-addressing mode is a valid candidate to be passed by reference; you do not have to limit yourself to static and local variables. For example, "[eax]" is a perfectly valid memory location, so you can pass this by reference (assuming you type-cast it, of course, to match the type of the formal parameter). The following example demonstrate a simple procedure with a pass by reference parameter:

```
program refDemo;

#include( "stdlib.hhf" )

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [eax]));

    end refParm;

    static
        i:int32:=5;

begin refDemo;

    stdout.put( "(1) i=", i, nl );
    mov( 25, i );
    stdout.put( "(2) i=", i, nl );
    refParm( i );
    stdout.put( "(3) i=", i, nl );


end refDemo;
```

The output produced by this code is

```
(1) i=5
(2) i=25
(3) i=12345
```

As you can see, the parameter *a* in `refParm` exhibits pass by reference semantics since the change to the value `a` in `refParm` changed the value of the actual parameter (`i`) in the main program.

Note that HLA passes the address of `i` to `refParm`, therefore, the *a* parameter contains the address of `i`. When accessing the value of the `i` parameter, the `refParm` procedure must deference the pointer passed in `a`. The two instructions in the body of the `refParm` procedure accomplish this.

Look at the code that HLA generates for the call to `refParm`:

```
pushd( &(i__hla_1884+0) );
call refParm__hla_1883;
```

(*i__hla_1884* is the back-end assembler compatible name that HLA generated for the static variable `i`.)

As you can see, this program simply computed the address of `i` and pushed it onto the stack. Now consider the following modification to the main program:

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

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [eax]));

    end refParm;

    static
        i:int32:=5;

    var
        j:int32;

begin refDemo;

    mov( 4, ebx );
    mov( 0, j );
    refParm( j );
    refParm( i[ebx] );
    lea( eax, j );
    refParm( [eax+ebx] );

end refDemo;
```

This version emits the following (pseudo-HLA syntax) code for the body of the main program:

```
// mov( 4, ebx );

mov( 4, ebx );

// mov( 0, j );

mov( 0, (type dword [ebp-8]) );

// refParm( j );
```

```
push( ebp );
add( -8, (type dword [esp]) );
call refParm__hla_1883;

// refParm( i[ebx] );

push( eax );
push( eax );
lea( i__hla_1884[ebx], eax );
mov( eax, [esp+4] );
pop( eax );
call refParm__hla_1883;

// lea( eax, j );

lea( [ebp-8], eax );

// refParm( [eax+ebx] );

push( eax );
push( eax );
lea( [eax+ebx*1], eax );
mov( eax, [esp+4] );
pop( eax );
call refParm__hla_1883;
```

As you can see, the code emitted for the last two calls is ugly. These calls would be good candidates for using the **call** instruction directly. Also see *Hybrid Parameters* elsewhere in this chapter. Another option is to use the **@use reg$_{32}$** option to tell HLA it can use one of the 32-bit registers as a scratchpad. Consider the following:

```
procedure refParm( var a:int32 ); @use esi;
        .
        .
        .
```

This sequence generates the following code (which is a little better than the previous example):

```
// mov( 4, ebx );

mov( 4, ebx );

// mov( 0, j );

mov( 0, (type dword [ebp-8]) );

// refParm( j );

lea( [ebp-8], esi );
push( esi );
call refParm__hla_1883;

// refParm( i[ebx] );

lea( i__hla_1884[ebx], esi );
push( esi );
call refParm__hla_1883;
```

```
// lea( eax, j );

lea( [ebp-8], eax );

// refParm( [eax+ebx] );

lea( [eax+ebx*1], esi );
push( esi );
call refParm__hla_1883;
```

As a rule, the type of an actual reference parameter must exactly match the type of the formal parameter. There are a couple exceptions to this rule. First, if the formal parameter is `dword`, then HLA will allow you to pass any four-byte data type as an actual parameter by reference to this procedure. Second, you can pass an actual `dword` parameter by reference if the formal parameter is a four-byte data type.

There is a third exception to the "the types must exactly match" rule. If the formal reference parameter is some data type HLA will allow you to pass an actual parameter that is a pointer to this type. Note that HLA will actually pass the *value* of the pointer, rather than the *address* of the pointer, as the reference parameter. This turns out to be convenient, particularly when calling Win32 API functions and other C/C++ code. Note, however, that this behavior isn't always intuitive, so be careful when passing pointer variables as reference parameters.

If you want to pass the value of a double word or pointer variable in place of the address of such a variable to a pass by reference, value/result, or result parameter, simply prefix the actual parameter with the **val** reserved word in the call to the procedure, e.g.,

```
refParm( val dwordVar );
```

This tells HLA to use the value of the variable rather than it's address.

You may also use the **val** keyword to pass an arbitrary 32-bit numeric value for a string parameter. This is useful in certain Win32 API calls where you pass either a pointer to a zero-terminated sequence of characters (i.e., a string) or a small integer "ATOM" value.

11.12 Untyped Reference Parameters

HLA provides a special formal parameter syntax that tells HLA that you want to pass an object by reference and you don't care what its type is. Consider the following HLA procedure:

```
procedure zeroObject( var object:byte; size:uns32 );
begin zeroObject;
    << code to write "size" zeros to "object" >
end zeroObject;
```

The problem with this procedure is that you will have to coerce non-byte parameters to a byte before passing them to `zeroObject`. That is, unless you're passing a byte parameter, you always have to call `zeroObject` thusly:

```
zeroObject( (type byte NotAByte), sizeToZero );
```

For some functions you call frequently with different types of data, this can get painful very quickly.

The HLA untyped reference parameter syntax solves this problem. Consider the following declaration of `zeroObject`:

```
procedure zeroObject( var object:var; size:uns32 );
begin zeroObject;
    << code to write "size" zeros to "object" >
```

```
end zeroObject;
```

Notice the use of the reserved word **var** instead of a data type for the object parameter. This syntax tells HLA that you're passing an arbitrary variable by reference. Now you can call `zeroObject` and pass any (memory) object as the first parameter and HLA won't complain about the type, e,g.,

```
zeroObject( NotAByte, sizeToZero );
```

Note that you may only pass untyped objects by reference to a procedure.

Note that untyped reference parameters always take the address of the actual parameter to pass on to the procedure, even if the actual parameter is a pointer. Normal pass by reference semantics in HLA will pass the value of a pointer, rather than the address of the pointer variable, if the base type of the pointer matches the type of the reference parameter. Sometimes you'll have the address of an object in a register or a pointer variable and you'll want to pass the value of that pointer object (i.e., the address of the ultimate object) rather than the address of the pointer variable. To do this, simply prefix the actual parameter with the **val** keyword, e.g.,

```
zeroObject( ptrVar );     // Passes the address of ptrVal
zeroObject( val ptrVar ); // Passes ptrVar's value.
```

11.13 Pass by Value/Result and Pass by Result Parameters

Although the behavior of pass by value/result and pass by result parameters is identical to pass by reference on a procedure call (that is, the caller passes the address of the object to the subroutine rather than the value), inside the procedure the behavior of these parameter-passing mechanisms is quite different. This section will discuss those differences and how you use pass by value/result and pass by result parameters in a program. The pass by result parameter-passing mechanism is actually a subset of the pass by value/result mechanism, so this section will fully describe the pass by value/result mechanism and then point out the difference between the two parameter-passing mechanisms.

One problem with the pass by reference calling sequence is that it is possible to create aliases of variables in a parameter list that lead to non-intuitive results in your program. Consider the following (very famous) example:

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

    procedure famous( var a:int32; var b:int32 );
    begin famous;

       // a := 5;

       mov( a, ebx );
       mov( 5, (type int32 [ebx]));

       // b := 10;

       mov( b, ebx );
       mov( 10, (type int32 [ebx] );

       // print a+b

       mov( a, ebx );
       mov( [ebx], eax );
       mov( b, ebx );
       add( [ebx], eax );
       stdout.put( "a+b=", (type int32 eax), nl );
```

```
    end famous;


static
   someVar:int32;

begin refDemo;

   famous( someVar, someVar );

end refDemo;
```

The output from this program is "a+b=20" which is somewhat counter-intuitive (the intuitive result, looking only at the code in *famous*, would be "a+b=15"). If you study the code above (no need to look at the low-level machine code), you'll discover the problem. In the call to *famous*, the main program passes the address of *someVar* in both parameter positions. Therefore, inside famous, both *a* and *b* contain the address of *someVar*. When *famous* stores the value 5 into the variable pointed at by *a*, it overwrites in *someVar* with 5. Because b also points at *someVar*, when *famous* stores the value 10 into the location pointed at by *b*, it overwrites the value 5 that it originally stored there. When *famous* accesses the values pointed at by *a* and *b* (to compute their sum), it retrieves the value 10 for both memory accesses and, therefore, computes the sum of 20.

The problem in this example is that *a* and *b* are aliases (different names for the same variable). The occurrence of aliases can sometimes create problems, as this example demonstrates. Note that the pass by value parameter-passing mechanism doesn't suffer from this problem because it makes a distinct copy of the parameter's value. Were you to use pass by value in the example above, you'd get the intuitive result of "a+b=15" because *a* and *b* would have both been separate variables in *famous'* activation record. The drawback to pass by value, of course, is that any modification to a pass by value parameter is not reflected in the actual parameter that was passed to the procedure.

The pass by value/result parameter-passing mechanism is a combination of the pass by value and pass by reference mechanisms: it provides a mechanism for passing values into and out of a procedure while avoiding the problem of aliases (by creating a local copy of the actual parameter's value in the activation record of the procedure). Here's how pass by value/result works:

the caller passes in an address of the actual parameter

the subroutine makes a copy of the actual parameter's data into a local (automatic) variable

the subroutine manipulate the copy of the data just as it would a value parameter or any other local (automatic) variable

before the subroutine returns, it copies the data from the local object to the memory location pointed at by the address that the caller passed to the procedure for the actual parameter.

Note that two copies of the data are made: one copy is made upon entry into the subroutine (from the actual parameter to the local copy) and one copy is made upon exit from the procedure (from the local copy to the actual parameter). Inside the procedure, however, all pass by value/result parameters have their own copy of their actual parameter's data, so there are no aliases. Were you to rewrite *famous* to use pass by value result, you'd get the intuitive result of "a+b=15".

The pass by value/result and pass by result parameter-passing mechanisms are unusual because their behavior is quite different when **@frame** or **@noframe** is specified for the procedure. Whenever you specify **@frame** (or if this is the default condition), then HLA will automatically allocate storage for the local copy of the data and *the formal parameter name will refer to that local data in the activation record*. You will not have direct access (via some sort of parameter name) to the address that the caller actually passed into the procedure. This is actually convenient and natural; you want to treat pass by value/result parameters as though they were simple values. Consider the following example:

```
program valresDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( valres a:int32; valres b:int32 );
```

```
    begin p;
    end p;


static
   someVar:int32;
   someVar2:int32;

begin valresDemo;

   p( someVar, someVar2 );

end valresDemo;
```

Here is the symbol table for procedure p in this code:

```
p               <0,proc>:Procedure type (ID=p__hla_1)
    --------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =8
    b               <1,var >:int32, (4 bytes, ofs:-8)
    a               <1,var >:int32, (4 bytes, ofs:-4)
    _parms_         <1,cons>:uns32, (4 bytes)  =8
    a               <1,vrp >:int32, (4 bytes, ofs:12)
    b               <1,vrp >:int32, (4 bytes, ofs:8)
    _finalize_      <1,val >:string, (0 bytes)  =""
    _initialize_    <1,val >:string, (0 bytes)  =""
    p               <1,proc>:
-----------------------------------
```

Note that the symbols *a* and *b* appear twice in *p's* symbol table. The first pair (which HLA always finds when searching through the symbol table) corresponds to the local copies of these parameters that HLA allocates on the stack. The second pair (which are inaccessible to your program because HLA always finds the other pair first) corresponds to the actual addresses that the caller pushes onto the stack.

Here is the code that HLA generates for procedure *p* (including the code it automatically generates to copy the value/result parameter data to the local copy):

```
procedure p__hla_1;
begin p__hla_1;

      // Set up the activation record:

       push( ebp );
       mov( esp, ebp );

      // Allocate storage for locals (specifically, for the
      // local copies of a and b):

       sub( 8, esp );

      // Copy the actual values pointed at by a and b to
      // the local copies:

       push( esi );
       push( ecx );
```

```
        mov( [ebp+12], esi );// Get pointer to a
        mov( [esi], ecx );// Get a's value
        mov( ecx, [ebp-4] );// Store value into local copy

        mov( [ebp+8], esi );// Get pointer to b
        mov( [esi], ecx );// Get b's value
        mov( ecx, [ebp-8] );// Store value into local copy

        pop( ecx );
        pop( esi );

// Exit from procedure p:

xp__hla_1__hla_:

       // Copy the data from the local copies back to
       // the actual parameters:

        push( edi );
        push( ecx );

        mov( [ebp+12], edi );// Get pointer to a
        mov( [ebp-4], ecx );// Get a's local value
        mov( ecx, [edi] );// Store into actual a

        mov( [ebp+8], edi );// Get pointer to b
        mov( [ebp-8], ecx );// Get b's local value
        mov( ecx, [edi] );// Store into actual b

        pop( ecx );
        pop( edi );

       // Clean up the activation record and leave:

        mov( ebp, esp );
        pop( ebp );
        ret( 8 );
end p__hla_1;
```

If the procedure has the **@noframe** option, then the formal parameter name refers to the address passed by the caller on the stack. It is your responsibility to allocate storage for the local copy of the pass by value/result parameter and copy the data from the address specified to your local copy. Here's the code above with the **@noframe** option:

```
program valresDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( valres a:int32; valres b:int32 ); @noframe;
    begin p;
    end p;


static
   someVar:int32;
   someVar2:int32;
```

```
begin valresDemo;

   p( someVar, someVar2 );

end valresDemo;
```

Here's *p's* symbol table for this code:

```
p               <0,proc>:Procedure type (ID=p__hla_1) parms:8
    --------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =0
    _parms_         <1,cons>:uns32, (4 bytes)  =8
    a               <1,vrp >:int32, (4 bytes, ofs:12)
    b               <1,vrp >:int32, (4 bytes, ofs:8)
    _finalize_      <1,val >:string, (0 bytes)  =""
    _initialize_    <1,val >:string, (0 bytes)  =""
    p               <1,proc>:
-----------------------------------
```

Note that there is only one set of *a* and *b* parameters in this symbol table and they refer to the addresses passed on the stack rather than to any local data. You are responsible for allocating the storage making a local copy of the data when the **@noframe** option is present. Essentially, when you have **valres** parameters in a procedure that has the **@noframe** option, the effect is the same as if you declared the parameters as pass by reference parameters.

Given the amount of code HLA generates to copy pass by value/result parameters to and from the actual parameter locations, you might question whether using pass by value/result is very efficient. As it turns out, if you're accessing the value/result objects frequently in a program, you can quickly recoup the cost of the data copy operation with the more efficient access to a local variable (versus the indirect access that takes place with a reference variable).

Pass by result parameters are very similar to pass by value/result. The only difference is that HLA-generated code (when **@frame** is active) does not copy any data into the local copy of the parameter variable. Pass by result parameters are more efficient than pass by value/result if you are only using such parameters to return a value to the caller.

Note that HLA uses callee-copying semantics for pass by value/result and pass by result parameters. It's also possible to use caller-copying semantics (though HLA doesn't directly support this). The way to achieve caller-copy semantics is to pass the parameters by value but not remove them from the stack upon return. When the procedure returns to the caller, the caller can pop the data off the stack and store it into the original actual parameter locations. Obviously, this scheme generates a lot more code if you call the procedure a large number of times, but it can be slightly more efficient in certain cases because the procedure won't need to preserve registers it uses to copy the data to and from the local copy of the parameter data.

11.14 Parameter Passing in HLA, Name and Lazy Evaluation Parameters

HLA provides a modicum of support for pass by name and pass by lazy evaluation parameters. A pass by name parameter consists of a thunk that returns the address of the actual parameter. A pass by lazy evaluation parameter is a thunk that returns the value of the actual parameter. Whenever you specify the **name** or **lazy** keywords before a parameter, HLA reserves eight bytes to hold the corresponding thunk in the activation record. It is your responsibility to create a thunk whenever calling the procedure.

There is a minor difference between passing a thunk parameter by value and passing a lazy evaluation or name parameter to a procedure. Pass by name/lazy parameters require an immediate thunk constant; you cannot pass a thunk variable as a pass by name or lazy parameter.

To pass a thunk constant as a parameter to a pass by name or pass by lazy evaluation parameter, insert the thunk's code inside "#{...}#" sequence in the parameter list and preface the

whole thing with the **thunk** reserved word. The following example demonstrates passing a thunk as a pass by name parameter:

```
program nameDemo;
#include( "stdio.hhf" );

    procedure passByName( name ary:int32; var ip:int32 );
    @nodisplay;
    const i:text := "(type int32 [ebx])";
    const a:text := "(type int32 [eax])";
    begin passByName;

        mov( ip, ebx );
        mov( 0, i );
        while( i < 10 ) do

            ary();  // Get address of "ary[i]" into eax.
            mov(i, ecx );
            mov( ecx, a );
            inc( i );

        endwhile;

    end passByName;

    procedure thunkParm( t:thunk );
    begin thunkParm;

        t();

    end thunkParm;


var
    index:int32;
    array:int32[10];
    th:thunk;

begin nameDemo;

    thunk th := #{ stdout.put( "Thunk Variable",nl ) }#;
    thunkParm( th );
    thunkParm( thunk #{ stdout.put( "Thunk Constant" nl ); }# );

    // passByName( th, index ); -- would be illegal;

    passByName
    (
        thunk
        #{
            push( ebx );
            mov( index, ebx );
            lea( eax, array[ebx*4] );
            pop( ebx );
        }#,
        index
    );
```

```
    mov( 0, ebx );
    while( ebx < 10 ) do

        stdout.put
        (
          "array[",
          (type int32 ebx),
          "]=",
          array[ebx*4],
          nl
        );
        inc( ebx );

    endwhile;

end nameDemo;
```

This program produces the following output:

```
Thunk Variable
Thunk Constant
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9
```

The main purpose of pass by name and pass by lazy evaluation parameters is to support deferred parameter evaluation. When you pass a value, reference, value/result, or result parameter to a procedure, the value of that parameter is evaluated exactly once, at the point of the procedure call. When you pass a pass by name or pass by lazy evaluation parameter to a procedure, you're passing a thunk that is called to evaluate the parameter's value whenever you want to reference that parameter's value. From HLA's perspective, pass by name and pass by lazy evaluation parameters are implemented exactly the same way. The intent is that the respective thunks for this parameters return the address (pass by name) of the object or the value (pass by lazy evaluation) of the actual parameter object.

11.15 Hybrid Parameter Passing in HLA

HLA's automatic code generation for parameters specified using the high-level language syntax isn't always optimal. In fact, sometimes it is downright inefficient (though, to be fair, the code generation has gotten much better over the past decade; there are only a few degenerate examples we can draw from today). This is because HLA makes very few assumptions about your program. For example, suppose you are passing a word parameter to a procedure by value. Since all parameters in HLA consume an even multiple of four bytes on the stack, HLA will zero extend the word and push it onto the stack. It does this using code like the following:

```
    pushw     0
    pushw     Parameter
```

Clearly, you can do better than this if you know something about the variable. For example, if you know that the two bytes following *Parameter* are in memory (as opposed to being in the next page of memory that isn't allocated, and access to such memory would cause a protection fault), you could get by with the single instruction:

```
push      dword ptr Parameter
```

Unfortunately, HLA cannot make these kinds of assumptions about the data because doing so could create malfunctioning code (actually, if *Parameter* is an automatic variable or a parameter passed in from some other call, then HLA will make this assumption - this is an example of how HLA's code generation has improved over the years).

One solution, of course, is to forego the HLA high-level language syntax for procedure calls and manually push all the parameters your self and call the procedure via the **call** instruction. However, this is a major pain involving lots of extra typing and produces code that is difficult to read and understand. Therefore, HLA provides a hybrid parameter passing mechanism that lets you continue to use a high-level language calling syntax yet still specify the exact instructions needed to pass certain parameters. This hybrid scheme works out well because HLA actually does a good job with most parameters (e.g., if they are an even multiple of four bytes, HLA generates efficient code to pass the parameters; it's only those parameters that have a weird size that HLA generates less than optimal code for).

If a parameter consists of the "#{" and "}#" brackets with some corresponding code inside the brackets, HLA will emit the code inside the brackets in place of any code it would normally generate for that parameter. So if you wanted to pass a 16-bit parameter efficiently to a procedure named "Proc" and you're sure there is no problem accessing the two bytes beyond this parameter, you could use code like the following:

```
Proc( #{ push( (type dword WordVar) ); }# );
```

Notice the similarity to pass by name/eval parameters. However, no **thunk** reserved word prefaces the code in this instance.

Whenever you pass a non-static[1] variable as a parameter by reference, HLA generates the following (MASM-syntax) code to pass the address of that variable to the procedure:

```
pusheax
pusheax
lea    eax, Variable
mov    [esp+4], eax
pop    eax
```

It generates this particular code to ensure that it doesn't change any register values (after all, you could be passing some other parameter in the EAX register). If you have a free register available, you can generate slightly better code using a calling sequence like the following (assuming EBX is free):

```
HasRefParm
(
   #{
          lea( ebx, Variable );
          push( ebx );
   }#
);
```

Note that HLA will generate slightly better code for automatic variables and parameters that don't have an indexed addressing mode attached to them. Examining the HLA output code (using -

1. Static variables are those you declare in the static, readonly, and storage sections. Non-static variables include parameters, VAR objects, and anonymous memory locations.

source and -hla command-line options) is a good way to see exactly what HLA is doing with your procedure calls.

11.16 Parameter Passing in HLA, Register Parameters

HLA provides a special syntax that lets you specify that certain parameters be passed in registers rather than on the stack. The following are some examples of procedure declarations that use this feature:

```
procedure a( u:uns32 in eax ); forward;
procedure b( w:word in bx ); forward;
procedure d( c:char in ch ); forward;
```

Whenever you call one of these procedures, the code that HLA automatically emits for the call will load the actual parameter value into the specified register rather than pushing this value onto the stack. You may specify any general-purpose 8-bit, 16-bit, or 32-bit register after the **in** keyword following the parameter's type. Obviously, the parameter must fit in the specified register. You may only pass reference parameters in 32-bit registers; you cannot pass parameters that are not one, two, or four bytes long in a register.

You can get in to trouble if you're not careful when using register parameters, consider the following two procedure definitions:

```
procedure one( u:uns32 in eax; v:dword in ebx ); forward;
procedure two( a:uns32 in eax );
begin two;

   one( 25, a );

end two;
```

The call to *one* in procedure *two* looks like it passes the values 25 and whatever was passed in for *a* in procedure *two*. However, if you study the HLA output code, you will discover that the call to *one* passes 25 for both parameters. They reason for this is because HLA emits the code to load 25 into EAX in order to pass 25 in the *u* parameter. Unfortunately, this wipes out the value passed into *two* in the *a* variable, hence the problem. Be aware of this if you use register parameters often.

11.17 Instruction Composition and Parameter Passing in HLA

You can use HLA's instruction composition feature in calls to HLA procedures. Consider the following simple example:

```
program instrCompDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( var a:dword ); @noframe;
    begin p;
    end p;

var
   someVar:int32;

begin instrCompDemo;

   p( [lea( eax, someVar)] );

end instrCompDemo;
```

You can also use instruction composition to compute certain expression values to pass as value parameters to an HLA procedure:

```
program instrCompDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( a:dword ); @noframe;
    begin p;
    end p;

var
    i   :int32;
    j   :int32;

begin instrCompDemo;

// Pass i+j as the value parameter:

p( add( mov( i, eax ), mov( j, ebx )) );

end instrCompDemo;
```

Here's the code that HLA generates for the procedure call in the last example:

```
mov( [ebp-8], eax );
mov( [ebp-12], ebx );
add( eax, ebx );
push( ebx );
call p__hla_1;
```

The HLA `returns` statement provides another mechanism for using instruction composition to pass parameters to an HLA function. Here is the basic syntax for the `returns` statement:

```
returns(  { << HLA statements >> }, "string" )
```

HLA will compile the statements between the braces and then return the string operand as the "returns" value for the entire construct. If this string contains an x86 register, a memory location, or an appropriate constant, HLA will emit the appropriate push instruction for that string operand if the `returns` statement appears as a procedure parameter. Here is an example of this usage:

```
// Pass i+j as the value parameter:

p
(
    returns
    (
        {
            mov( i, eax );
            mov( j, ebx );
            add( eax, ebx );
        },
        "ebx"
    )
);
```

Note that this example emits the same code as the previous example. Though this example is a bit longer, it’s also much easier to read and comprehend.

11.18 Lexical Scope

HLA is a block-structured language that enforces the scope of local identifiers. HLA uses lexical scope to determine when and where an identifier is visible to the program. Identifiers declared within a procedure are always visible within that procedure and to any local procedures declared after the identifier. Local identifiers are never visible outside the procedure declaration. The scoping rules are similar to languages like Pascal, Ada, and Modula-2. As an example, consider the following code:

```
program scopeDemo;

#include( "stdio.hhf" );

var
    i:int32;
    j:int32;
    k:int32;

    procedure lex1;
    var
        i:int32;
        j:int32;

        procedure lex2;
        var
            i:int32;
        begin lex2;

            mov( i, eax ); //1
            mov( ebx::j, eax ); //2
            mov( ecx::k, eax ); //3

        end lex2;

    begin lex1;

        mov( i, eax );          //4
        mov( j, eax );          //5
        mov( ecx::k, eax );      //6

    end lex1;


    procedure alsolex1;
    var
        i:int32;
        m:int32;
    begin alsolex1;

        mov( i, eax );          //4
        mov( m, eax );          //5
        mov( ecx::k, eax );     //6

    end alsolex1;
```

```
begin scopeDemo;

    mov( i, eax );              //7
    mov( j, eax );              //8
    mov( k, eax );              //9

end scopeDemo;
```

Note: the purpose of the ebx:: and ecx:: prefixes on certain variables will become clear in a moment. Also note that this code is not functional, it was written only as an illustration.

In this example, you will note that `lex2` is nested within `lex1`, which is nested within the main program. The `alsolex1` procedure is nested within the main program but inside no other procedure. To describe this arrangement, compiler writers use the term *lex level* to denote the depth of nesting. HLA defines the main program to be *lex level zero*. Each time you nest a procedure you increase its lex level. So `lex1` is at lex level one since it is directly nested inside the main program at lex level zero. The `lex2` procedure is at lex level two because it is nested inside the `lex1` procedure. Finally, `alsolex1` is also at lex level one because it is nested inside the main program (which is lex level zero).

Within a given procedure (or the main program), all identifiers must be unique. That is, you cannot have two symbols named *i* in the same procedure. In different procedures, however, you may reuse the names. If all procedures were written at lex level one, then no procedure would be able to directly access the local variables in any other procedure (this is the case with the C/C++ language). In block-structured languages, like HLA, it is possible to access certain non-local variables in other procedures if the current procedure (whose code is attempting to access said variable) is nested within the other procedure.

In the example above, `lex2` accesses three variables: `i`, `j`, and `k`. The `i` variable is local to `lex2`, so there is nothing surprising here. The `j` variable is local to `lex1` and global to `lex2`. Likewise, the `k` variable is global to both `lex1` and `lex2` yet `lex2` can access it. Whenever a procedure is nested within another (either directly or indirectly), the nested procedure can access all variables in the global, nesting, procedures (including the main program)[1] unless the procedure declares a local name with the same name as a global name (the local name always takes precedence in this case). The term "scope" refers to the visibility of these names.

Being able to use a name during compilation is one thing, accessing the memory location associated with that name at run-time is something else entirely. Most block-structured high-level languages (HLLs) emit lots of extra code to access these "intermediate" and global variables for you. Why the extra code? Well remember, local procedure variables are accessed on the stack by indexing off the EBP register (which points at a procedure's "activation record"). When a procedure like `lex1` above calls a local procedure like `lex2`, the `lex2` procedure promptly saves the value in EBP (that points at `lex1`*'s* activation record) and it points EBP at the new activation record for `lex2`. Unfortunately, `lex2` no longer has access to `lex1`*'s* local variables since EBP no longer points at `lex1`*'s* locals. This creates a bit of a problem.

"But wait!" you exclaim. "EBP is pointing at the pointer to `lex1`*'s* activation record, why not just use double indirection to get the pointer to `lex1`*'s* locals?" This is a good idea, but it fails if `lex2` is recursive. There are two or three general solutions to this problem; HLA uses a *display* to access non-local values.

A display is nothing more than an array of pointers. `Display`*[0]* is a pointer to the most recent activation record at lex level zero, `Display`*[1]* is a pointer to the most recent activation record at lex level one, `Display`*[2]* is a pointer to the most recent activation record at lex level two, etc. (note the use of the phrase *most recent*. This ensures that displays work properly even when recursion occurs). With a display, to access a non-local variable, you just go to the memory location specified by `Display`*[varlex]* + *varoffset* where `varlex` is the lex level of the symbol you wish to access and `varoffset` is the offset into the activation record where the variable's data can be found.

Sound complex? Actually, HLA simplifies this quite a bit. First, as long as you don't specify the `@nodisplay` procedure option, HLA automatically emits the code to build a display at the

1. Strictly speaking, this isn't true. The nested procedure has access to all global variables that were declared before the procedure's declaration.

start of the procedure's code[1]. HLA also defines a run-time variable, `_display_`, that points at (the end of) this array of pointers. To access a non-local variable requires two instructions, one to fetch the address of the variable's activation record and one to access the variable. Correcting the previous program, the code would look something like this:

```
procedure lex2;
var
    i:int32;
begin lex2;

    mov( i, eax );

    // access non-local variable j
    // at lex level 1.

    mov( _display_[-1*4], ebx );
    mov( ebx::j, eax );

    // access non-local variable k
    // at lex level 0.

    mov( _display_[0], ecx );
    mov( ecx::k, eax );

end lex2;
```

There are two things to note about the display: first, the entries are stored at negative indices in the array (0, -1, -2, etc) rather than at positive indices (this is due to Intel's implementation of displays). Second, don't forget that this is a run-time array of double-words so you must multiply each index by the array element size, which is four in this case.

Once you've loaded the address into a register, the reg::var syntax tells HLA to use the specified register rather than EBP as the pointer to the variable's activation record. The "mov(ecx::k,eax);" instruction, for example, compiles to "mov eax, [ecx+koffset]" where `koffset` represents the offset of `k` in the main program's activation record.

In general, few programs take advantage of nested procedures and access to local variables, so it is very common to find programmers putting `@nodisplay` after all their procedures. Of course, if you do this, HLA does not generate display and access to non-local variables (declared in the **var** section) is not possible. Of course, static variables are not allocated in the activation record, so you always have access to non-local static variables even if you don't generate the code for a display.

1. It is important that all nested procedures construct the display. You couldn't use the `@nodisplay` option in lex1 and expect lex2 to properly build the display. In general, unless you know exactly what you are doing, your procedures should all have the `@nodisplay` option, or none of them should have it.