10 HLA Program Structure and Organization

10.1 HLA Program Structure

HLA supports two types of compilations: *programs* and *units*. A *program* is an HLA source file that includes the code (the "main program") that executes immediately after the operating system loads the program into memory. A *unit* is a module that contains procedures, methods, iterators, and data that is to be linked with other modules. Note that units may be linked with other HLA modules (including an HLA main program) or with code written in other languages (including high-level languages or other x86 assembly languages). This chapter will discuss the generic form of an HLA program; see the chapter on *HLA Units and External Compilation* for a detailed description of HLA units.

An executable file must have exactly one main program (written either in HLA or some other language). Therefore, most applications written entirely in HLA will have exactly one program module and zero or more units (it is possible to fake a program module using units; for more information see the on-line documents "Taking Control Over Code Emission" and "Calling HLA Code from Non-HLA Programs with Exception Handling" on Webster (http://webster.cs.ucr.edu). Therefore, the best place to begin discussion HLA program structure is by defining the HLA **program**. Here's the minimalist HLA program:

```
program pgmId;
begin pgmID;
end pgmID;
```

In this example, *pgmID* is a user-defined identifier that names the program. Note that this name is local to the program (that is, it is not visible outside the source file and neither the source file's name nor the executable's file name need be the same as this name (though it's not a bad idea to make them the same). Note that the exact same identifier following the **program** reserved word must follow the **begin** and **end** reserved words.

The minimalist HLA program, above, doesn't do much; if you compile and execute this program it will immediately return control to the operating system. However, this short program actually does quite a bit for an empty assembly language program. When you create an HLA program, you're asking HLA to automatically generate some template code to do certain operations such as initializing the exception-handling system, possibly setting up command-line parameters for use by the program, and emitting code to automatically return control to the operating system when the program completes execution (by running into the **end** *pgmID* clause). As this code is generally needed for every HLA assembly language program, it's nice that the HLA compiler will automatically emit this template code for you. If you happen to be a die-hard assembly programmer and you don't want the compiler emitting any instructions you haven't explicitly written, fear not, HLA doesn't force you to accept the code it's written; for more details, see the "Taking Control Over Code Emission" article on Webster that was mentioned earlier.

A non-minimalist HLA program takes the following generic form:

```
program pgmId;
    << declarations >>
begin pgmID;
    << main program instructions>>
end pgmID;
```

The <<*declarations*>> section is where you will put the declarations/definitions for constants, data types, variables, procedures, methods, iterators, tables, and other data. The << *main program instructions* >> section is where you will put machine instructions and HLA HLL-like statements.

An HLA unit is even simpler than an HLA program. It takes the following form:

```
unit unitId;
    << declarations >>
end unitID;
```

In this example, *unitID* is a user-defined identifier that names the unit. Note that this name is local to the unit (that is, it is not visible outside the source file and the source file's name need be the same as this name. Note that the exact same identifier following the **unit** reserved word must follow the **end** reserved word. Unlike programs, units do not have a **begin** clause following by a sequence of instructions; this is because units don't provide the main program code for the application. Again, for more details about units, see the chapter on HLA Units.

10.2 The HLA Declaration Section

The declaration section in an HLA program or unit is relatively complex, supporting the definition and declaration of most of the components in the HLA program or unit. An HLA declaration section generally contains one or more of the following items:

- A labels section (**label**)
- A constant declaration section (
- **const**)
- A values declaration section (
- **value**)
- An automatic variables declaration section (
- **var**)
- An initialized static data storage declaration section (
- **static**)
- An initialized read-only data storage declaration section (
- **readonly**)
- An uninitialized static data storage declaration section (
- **storage**)
- A procedures declaration section (
- **proc**)
- Old-style procedure, method, and iterator declarations
-
- A
- **namespace** declaration section

These sections may appear in any order in a **program** or **unit** declarations section and multiple instances of each of these sections may appear in the declarations. The following subsections describe each of these declaration sections in detail.

10.2.1 The HLA LABEL Declaration Section

The HLA label section is a very special-purpose (and rarely used) declaration section in which you declare forward-referenced and external statement labels. The syntax for the **label** section is either of the following:

```
label
   << label declarations >>

or

label
   << label declarations >>
endlabel;
```

The **endlabel** clause is optional. If it is present it explicitly marks the end of the forward label declaration section; if it is absent, then the next declaration section or the **begin** keyword will implicitly end the forward label declaration section.

Each label declaration takes one of the three following forms:

```
userLabel_1;
userLabel_2; external;
userLabel_3; external( "externalLabelName" );
```

In these examples, *userLabel_*x (x=1, 2, or 3) is a user-defined identifier.

The first example above is a *forward label declaration*. This tells HLA that you're promising to declare the statement label within the scope of the **label** section (HLA will generate an error if you fail to declare the statement label within the scope of the **label** statement).

The *scope* of a **label** statement is the body of instructions associated with the main program, procedure, method, or iterator that immediately contains the **label** declaration section. For example, if the label statement appears in the declaration section of an HLA program, the corresponding statement label must be defined in the body of that program:

```
program labelDemo;
label
    someLabel;

    << other declarations >>
begin labelDemo;

    << main program instructions, part 1 >>
    someLabel: // someLabel must be defined in this code.
    << main program instructions, part 2 >>

end labelDemo;
```

Note that HLA automatically handles forward-referenced labels within the (machine instructions) body of a program, procedure, method, or iterator, without an explicit **label** declaration. The following is legal even though you do not have a forward declaration of *someLabel*:

```
program labelDemo;
    .
    .
    .
begin labelDemo;


            .
            .
            .
        lea( eax, &someLabel );
            .
            .
            .
        jmp someLabel;
            .
            .
            .
    someLabel: // someLabel's declaration appears after its use.
```

```
            .
            .
            .

end labelDemo;
```

The above is legal because the procedure references *someLabel* in the same scope where it is declared. Now consider the following example:

```
program labelDemo;
    .
    .
    .
    procedure ReferencesSomeLabel;
        .
        .
        .
    begin ReferencesSomeLabel;
            .
        .
            .
        lea( eax, &someLabel );// Illegal! someLabel is not defined in this
procedure.
            .
            .
            .
    end ReferencesSomeLabel;

begin labelDemo;


            .
            .
            .
    someLabel: // someLabel's declaration appears outside the scope of its
use.

            .
            .
            .

end labelDemo;
```

HLA will generate an error in this example because forward references to statement labels must be resolved within the scope of the procedure (or program) containing the forward reference. When HLA encounters the "**end** *ReferencesSomeLabel*;" clause in the procedure above, it will report that you haven't defined *someLabel* in that procedure. The solution to this problem is to use the **label** statement to create a forward symbol definition so that *someLabel* is defined (albeit at a different lex level) when HLA encounters the **lea** statement in the previous example. The following code demonstrates how to do this:

```
program labelDemo;
label
    someLabel;
    .
    .
```

```
    .
    procedure ReferencesSomeLabel;
        .
        .
        .
    begin ReferencesSomeLabel;
            .
        .
            .
        lea( eax, &someLabel );// This is legal because of the label
statement.
            .
            .
            .
    end ReferencesSomeLabel;

begin labelDemo;


            .
            .
            .
    someLabel: // someLabel had a forward declaration.

            .
            .
            .

end labelDemo;
```

You can also create external label definitions by attaching the **external** option to a label definition. External label definitions take one of two forms:

```
label
    someLabel; external;
    someExtLabel;external( "externalName" );
```

The first form assumes that *someLabel* is defined (and the name is made public) in some other source/object module using the name *someLabel*. The second form assumes that "*externalName*" is defined in some other source/object module and uses the name *someExtLabel* to refer to that symbol.

To create a public label that you can reference in another source module, you put an external label definition in the same source file as the actual symbol declaration, e.g.,

```
program labelDemo;
label
    someLabel; external;
    .
    .
    .

begin labelDemo;


            .
            .
```

```
            .
    someLabel: // someLabel is a public symbol.

            .
            .
            .

end labelDemo;
```

The **label** statement rarely appears in most HLA programs. It is very unusual to reference a symbol that is declared outside the scope of that usage. External symbols are usually procedures, methods, or iterators, and a program will typically use an external **procedure**, **iterator**, or **method** declaration rather than a **label** statement to declare such symbols. Nevertheless, **label** declarations are necessary on occasion, so you should keep the forward label declaration statement in mind.

Note that **label** declarations will not make a local symbol in some scope (that is, within some procedure) visible to code outside that scope. The following will generate an error:

```
program labelDemo;
label
    someLabel;
    .
    .
    .
    procedure declaresSomeLabel;
        .
        .
        .
    begin declaresSomeLabel;
            .
        .
            .
        someLabel:// This is local to this procedure.
            .
            .
            .
    end declaresSomeLabel;

begin labelDemo;


            .
            .
            .
    // This does not reference someLabel in declaresSomeLabel!

    lea( eax, &someLabel );

            .
            .
            .

end labelDemo;
```

The scope of the symbol *someLabel* defined in *declaresSomeLabel* is limited to the *declaresSomeLabel* procedure. In order to make *someLabel* visible outside of *declaresSomeLabel*, you must make that symbol global. This is done by following the label declaration with two colons instead of one colon:

```
program labelDemo;
    .
    .
    .
    procedure declaresSomeLabel;
        .
        .
        .
    begin declaresSomeLabel;
            .
        .
            .
        someLabel::// This is a global symbol.
            .
            .
            .
    end declaresSomeLabel;

begin labelDemo;


            .
            .
            .
    // This is legal

    lea( eax, &someLabel );

            .
            .
            .

end labelDemo;
```

Note that global symbols are not automatically public. If you need a symbol to be both global to a procedure and public (visible outside the source file), you must also define that global symbol as external in a **label** statement:

```
program labelDemo;
label
    someLabel; external;
    .
    .
    .
    procedure declaresSomeLabel;
        .
        .
        .
    begin declaresSomeLabel;
            .
        .
            .
        someLabel::// This is a global and public symbol.
            .
            .
```

```
            .
        end declaresSomeLabel;

    begin labelDemo;


            .
            .
            .
        // This is legal

        lea( eax, &someLabel );

            .
            .
            .

    end labelDemo;
```

Note that global label declarations only make the symbol global at the previous lex level, not across the whole program. The following will not work properly because *label1* is only visible in the *q* and *p* procedures, not in the main program.

```
program t;
label
    label1;

procedure p;

    procedure q;
    begin q;

        label1::

    end q;

begin p;
end p;


begin t;

    lea( eax, label1 );

end t;
```

The solution to this problem is to make the symbol public by declaring it **external** in both the *q* procedure and in the main program:

```
program t;
label
    label1; external;

procedure p;
```

```
    procedure q;
    label
       label1; external;

    begin q;

       label1:

    end q;

begin p;
end p;


begin t;

    lea( eax, label1 );

end t;
```

Of course, referencing a label in a nested procedure like this is highly unusual and is probably an indication of a poorly designed program. If you find yourself writing this kind of code, you might want to reconsider your program's architecture.

10.2.2 The HLA CONST Declaration Section

The HLA **const** section is where you declare symbolic (manifest) constants in an HLA program or unit. The syntax for the **const** section is either of the following:

```
const
    << constant declarations >>

or

const
    << constant declarations >>
endconst;
```

The **endconst** clause is optional at the end of the constant declarations in a declaration section; some programmers prefer to explicitly end a constant declaration section with **endconst**, others prefer to implicitly end the constant declarations (by starting another declaration section or with the **begin** keyword). The choice is yours, the language doesn't prefer either method nor does good programming style particularly specify one syntax over the other.

Each constant declaration takes one of the following forms:

```
userDefinedID := <<constant expression>>;
or
userDefinedID : typeID := <<constant expression>>;
```

Here are some examples:

```
const
    hasEdge_c := false;
    elementCnt_c:= 25;
```

```
    weight_c:= 32.5;

    debugMode_c:boolean:= true;
    maxCnt_c:uns32:= 15;
    oneHalf_c:real32:= 0.5;
```

The "_c" suffix is an HLA programming convention that tells the reader the identifier is a constant identifier. Although it's probably good programming style for you to follow this convention in your own HLA programs, HLA does not require this suffix on constant identifiers; any valid HLA identifier is fine when creating symbolic constants.

If you do not specify a data type for the symbolic constant declaration (as the first three examples above demonstrate), then HLA will infer the data type from the type of the constant expression. While this is convenient in many cases, do be aware that HLA might not choose the same data type you would explicitly provide. This is because a constant expression's type can be ambiguous, in which case HLA will use whatever type it finds convenient that will work. In the examples above, *hasEdge_c* must be a boolean constant because there is no ambiguity about the type of the constant **false**. The remaining two examples without an explicit type (*elementCnt_c* and *weight_c*) do not have constant expressions with an unambiguous type. The constant 25 is valid for types **uns8, uns16, uns32, uns64, uns128, byte, word, dword, qword, tbyte, lword, int8, int16, int32, int64,** and **int128** (and even **real32**, **real64**, and **real80** if you really want to push things). The HLA language definition does not require this constant (25) to assume any one of these particular values; HLA is free to choose whatever compatible type it wants for this constant. In most cases, it won't make a difference whether HLA chooses the type **uns8** or **uns32** for this constant (or any other of the legal types). However, there are many times that HLA might choose a type that will create problems with the code you're writing; therefore, it's a good idea to always explicitly provide a data type as do the last three examples above.

Note that constant expressions in a constant declaration support all the valid constant expression types discussed in the chapter on HLA Language Elements, including string, character set, array, record, union, and pointer constants. Indeed, it is often more convenient to create a constant for some structured data type and use that constant when initializing a static object than to assign the structured constant directly to the static object, e.g.,

```
const
    myArray_c :dword[ 8 ]:= [0,1,2,3,4,5,6,7];

static
    myArray:dword[@elements(myArray_c)] := myArray_c;
```

Note the use of *@elements(myArray_c)* rather than *8* to specify the number of dword array elements in the *myArray* declaration. By declaring the static array this way, you can change the *myArray_c* constant declaration by adding or removing array elements and the declaration for *myArray* will adjust its size automatically when you recompile.

One benefit to use structured constant declarations to initialize static objects is that you have full access to the (individual element or field) values of that structured constant during assembly. For example, you could reference *myArray_c[0]* in an HLA compile-time language sequence and know that you're getting the same value that goes into element zero of *myArray* at run time.

Objects you declare in a **const** section, as the name suggests, have a fixed value throughout the scope containing that symbol declaration. The value remains fixed both at compile time and at run time. Note, however, that HLA supports block-structured scoping rules so the symbol and its value might not be visible or available for use outside the scope in which you've declared the symbol. Consider the following program example:

```
program constDemo;
const
    sym := 10;
    .
    .
    .
    procedure usesSym;
```

```
    begin usesSym;
            .
        .
            .
        mov( sym, eax );  // Loads 10 into eax
            .
            .
            .
    end usesSym;

begin constDemo;


            .
            .
            .
mov( sym, eax ); // Loads 10 into eax
            .
            .
            .

end constDemo;
```

In this example, both instructions that use the symbol sym reference the same object and load the same value into eax. This is because HLA's block-structured scoping rules make global symbols visible inside procedures that don't redefine the symbol. Consider, however, the following example:

```
program constDemo;
    .
    .
    .
    procedure usesSym;
    const
        sym := 10;
    begin usesSym;
            .
        .
            .
        mov( sym, eax );  // Loads 10 into eax
            .
            .
            .
    end usesSym;

begin constDemo;


            .
            .
            .
mov( sym, eax ); // This is illegal!
            .
            .
            .
```

```
end constDemo;
```

HLA will reject this example because the second usage of *sym*, in the main program, is outside the scope of the symbol's declaration within the *usesSym* procedure (see the chapter on procedures for more information about HLA's scoping rules). Now consider this last example:

```
program constDemo;
const
    sym = 10;
    .
    .
    .
    procedure usesSym;
    begin usesSym;
            .
        .
            .
        mov( sym, eax );  // Loads 10 into eax
            .
            .
            .
    end usesSym;

    procedure declaresLocalSym;
    const
        sym := 25;
    begin declaresLocalSym;
            .
        .
            .
        mov( sym, eax );  // Loads 25 into eax
            .
            .
            .
    end declaresLocalSym;

begin constDemo;


            .
            .
            .
mov( sym, eax ); // Loads 10 into eax
            .
            .
            .

end constDemo;
```

This example seems to contradict the statement given earlier that constant declarations can have only a single value throughout the source file. The first usage of *sym* in the *usesSym* procedure loads the value 10 into EAX, just as in the earlier example. In the procedure *declaresLocalSym* we see a second declaration of *sym* with the value 25. When the code in this procedure references *sym*, HLA substitutes the value 25 for the symbol. This action seems to contradict the statement that a constant symbol has a fixed value throughout the compilation. However, the second declaration of sym is not a redeclaration of the original symbol (giving it a new value); instead, this is the declaration of a brand-new symbol that just happens to share the

same name (*sym*) as a symbol declared in the main program. The scope of this second symbol is limited to the procedure in which it is defined (and any procedures declared within it, though there are no such procedures in this example). The original symbol is not redefined, it is simply "hidden from view." At the end of *declaresLocalSym*, its local symbols are hidden and the global symbols are again visible. Note that the constant *sym* reverts to the value 10 at this point. Again, not meaning to sound redundant, it's important for you to understand that the two *sym* identifiers represent different constant objects whose visibility is controlled by the scope of those identifiers. The chapter on Procedures goes into greater detail about scope and how it affects the visibility of your symbols.

Unlike labels, you cannot create "external" constants. HLA **const** objects are *manifest* constants. This means that HLA substitutes the values of the **const** symbols wherever they appear in the source file before actually compiling the statements containing those symbols. In the object code file that HLA produces, the constant symbol no longer exists in any form; just the value of that symbol (unlike, say, an external label or procedure definition that passes the name of the symbol on to the linkage editor [linker] in order to properly combine object modules containing mutually-dependent symbols). If you want to use a constant symbol in multiple source files, the appropriate way to do this is to put the symbol into a header file and include that header file in all the source files that use the symbol.

10.2.3 The HLA VAL Declaration Section and the Compile-Time "?" Statement

The HLA **val** declaration section is very similar to the **const** declaration section insofar as you use it to declare symbol names that have a constant value at run time. The difference between the **val** and the **const** declaration sections is that you can reassign a different value to a **val** constant during the assembly/compilation process. This is useful for creating compile-time variables and handling a few other situations where **const** objects won't work. The syntax for the **val** section is either of the following:

```
val
    << value declarations >>
```

or

```
val
    << value declarations >>
endval;
```

As for the **const** section, either syntax is perfectly acceptable to HLA and either form neither form is particular preferred based on good programming style.

The syntax for the individual value declarations is identical to that of the constant declarations in a **const** section. There are two main differences: simple declarations without an assignment and value redefinitions.

The first difference is that a value declaration may consist of a constant identifier and a type identifier without the assignment of a constant expression. For example:

```
val
    sym :uns32;
```

This form creates the identifier without giving it an explicit value. HLA assumes that you are going to assign a value to this **val** constant before you use the value of that constant.

The second difference is that you can redefine the value of a value object multiple times in a program, for example:

```
program valDemo;
val
    sym :uns32;
```

```
        .
        .
        .
val
    sym := 10;
        .
        .
        .
val
    sym := sym + 1;
        .
        .
        .

begin valDemo;


            .
            .
            .
mov( sym, eax ); // Loads 11 into eax
            .
            .
            .

end valDemo;
```

Note that value redefinition in a **val** section only takes place when reassigning the value in the same scope as the original symbol definition. If you attempt to redefine the symbol at some point in the program that would have a different scope, then you will simply create a new object with the same name that is limited to the scope of the new definition. For example, consider the following code:

```
program t;
val
    i:=0;
endval;

    procedure u;
    val
        i := 1;
    begin u;

        #print( "i=", i )

    end u;

begin t;

    #print( "i=", i );

end t;
```

This example prints "i=1" and then "i=0" during compilation. The second declaration of *i* in procedure *u* is a local symbol (local to *i*), this declaration does not affect the original value of the *i* constant. To overcome this problem and provide a way to reassign the value of a **val** constant

anywhere in an HLA source file (including outside **val** declaration sections), HLA provides the compile-time assignment statement. An HLA compile-time assignment statement is legal anywhere a space is legal within the confines of an HLA **program** or **unit**. The HLA compile-time assignment statement takes one of the following two forms:

```
?valIdentifier := <<constant expression>>;
?valIdentifier :typeID := <<constant expression>>;
```

In these examples *valIdentifier* is either an undefined symbol or a constant identifier that was previously declared in a **val** declaration section or an HLA "?" compile-time assignment statement. In some respects, the HLA compile-time assignment statement is more flexible than the assignment of a value constant within a **val** section. Consider the following two programs that produce identical results:

```
program t1;
val
    i:=10;
begin t1;

    #print( "i=", i );  // Prints "10" at compile-time

end t1;


program t2;
?i:=10;
begin t2;

    #print( "i=", i );  // Prints "10" at compile-time

end t2;
```

There is, however, a major limitation to defining **val** constant identifiers in an HLA compile-time assignment statement: you cannot redefine the meaning of a symbol within some different scope (at a higher lex level) when using the compile-time assignment statement. For example, the following is illegal:

```
program t;
static
    i:uns32;

    procedure u;
    ?i := 1;// This is illegal!
    begin u;

        #print( "i=", i )

    end u;

begin t;
    .
    .
    .
end t;
```

The problem here is that the HLA compile-time assignment statement only defines a new symbol if it was previously undefined. In this example the symbol *i* was already defined as a static

variable. As only value constant identifiers may appear in an HLA compile-time assignment statement, HLA will reject this program. Note, however, that the following is legal:

```
program t;

    procedure u;
    ?i := 1;// This is legal!
    begin u;

        #print( "i=", i )

    end u;

static
    i:uns32;

begin t;
    .
    .
    .
end t;
```

This program will compile (assuming you have something reasonable between the **begin** and **end** clauses) and print "i=1" during compilation. The difference here is that *i* was undefined at the point of the "?i := 1;" assignment statement so HLA was able to create a constant identifier (local to procedure *u*). At the end of procedure *u*, the symbol *i* was hidden from the rest of the compilation so the declaration of *i* in the main program does not produce a duplicate definition error. By the way, if you really needed to define *i* as a value constant with procedure *u* in the illegal example, you could do the following:

```
program t;
static
    i:uns32;

    procedure u;
    val
        i:uns32 := 1;
    begin u;

        #print( "i=", i )

    end u;

begin t;
    .
    .
    .
end t;
```

Value constants you declare in a **val** section are not subject to the restriction that the symbol must be (globally) undefined at the point of the declaration. In the example above, *i* is a local symbol in procedure *u* that just happens to be a **val** object with the value one.

Although **val** objects are syntactically similar to const objects, you use them in an HLA program in almost completely different ways. The main purpose for **val** objects is to create compile-time variables that you can use to control compilation via compile-time loops, conditional compilation, and macros. Comparing **const** and **val** objects at compile time is quite similar to

comparing **readonly** and **static** objects at run time. The following example demonstrates how you can use a **val** object in a program to unroll a loop at compile time:

```
program unroll;
static
   ary:uns32[16];

begin t;

   ?loopIndex :uns32 := 0;
   #while( loopIndex < 16 )

      mov( loopIndex, ary[ loopIndex*4 ] );
      ?loopIndex := loopIndex + 1;

   #endwhile
      .
      .
      .
end t;
```

Note that the **#while** loop executes at compile time, not at run time. The code between the **#while** and **#endwhile** compile-time statements is equivalent to the following 16 statements:

```
      mov( 0, ary[ 0*4 ] );
      mov( 1, ary[ 1*4 ] );
      mov( 2, ary[ 2*4 ] );
      mov( 3, ary[ 3*4 ] );
      mov( 4, ary[ 4*4 ] );
      mov( 5, ary[ 5*4 ] );
      mov( 6, ary[ 6*4 ] );
      mov( 7, ary[ 7*4 ] );
      mov( 8, ary[ 8*4 ] );
      mov( 9, ary[ 9*4 ] );
      mov( 10, ary[ 10*4 ] );
      mov( 11, ary[ 11*4 ] );
      mov( 12, ary[ 12*4 ] );
      mov( 13, ary[ 13*4 ] );
      mov( 14, ary[ 14*4 ] );
      mov( 15, ary[ 15*4 ] );
```

This is because each iteration of the **#while** loop at compile time compiles all of the statements between the **#while** and **#endwhile** statements. For more information on the **#while/#endwhile** statement and using **val** objects as compile-time variables, please see the chapter on the HLA Compile-Time Language.

10.2.4 The HLA TYPE Declaration Section

Examples of the HLA **type** declaration section have been so numerous in the chapter on HLA Language Elements that describing them here is almost redundant (please review that chapter for more details). Nevertheless, for completeness and for the sake of a reference guide, this section describes the syntax of a **type** declaration section. A type declaration section takes one of the following two forms:

```
type
   << type declarations >>
```

```
or

type
    << type declarations >>
endtype;
```

A type declaration takes one of the following forms:

```
newTypeID : typeID;
newTypeID : typeID [ list_of_array_dimensions ];
newTypeID : procedure (<<optional_parameter_list>>);
newTypeID : record  <<record_field_declarations>> endrecord;
newTypeID : union  <<union_field_declarations>> endunion;
newTypeID : class  <<class_field_declarations>> endclass;
newTypeID : pointer to typeID;
newTypeID : enum{ <<list_of_enumeration_identifiers>> };
```

The purpose of the HLA type section is to declare a new type identifier that you can use when declaring **const, val, var, static, readonly,** and **storage** objects. You can also use type identifiers you declare in an HLA **type** section to define procedure prototypes in an HLA **proc** section. Each of the forms above deserves its own subsection to describe it, so the following subsections do just that.

Note that a type declaration only defines a type identifier you can use for declaring other objects in an HLA source file. A type declaration does not create a variable or constant object of the specified type. You can use the type identifier in some other declaration section (**const, val, var, static, readonly, storage,** etc.) to actually define an object of that type.

10.2.4.1 typeID

Before describing the valid syntax forms for the type declaration section, it's worthwhile to take a moment to describe the *typeID* item that appears in many of the type declarations. The *typeID* item is a single identifier whose classification is "type" (duh). This can be any of the HLA built-in types:

```
boolean
enum
uns8
uns16
uns32
uns64
uns128
byte
word
dword
qword
tbyte
lword
int8
int16
int32
int64
int128
char
wchar
real32
real64
```

```
real80
real128
string
zstring
u
nicode
cset

text
thunk.
```

The *typeID* can also be any user-defined type identifier you've previously declared in a type declaration section.

10.2.4.2 newTypeID : typeID;

The least complex type declaration is a simple type isomorphism, where you take an existing type and create a new type with all the same attributes except that you use a different name for the type. For example, suppose you want to use the identifier *integer* rather than **int32** in your programs. You could do this with the following type declaration:

```
type
    integer :int32;
```

Within the scope of this declaration, you can use the type name *integer* anywhere you want to declare a 32-bit signed integer object.

Warning: exercise care when using type isomorphisms of built-in types in an HLA program. If you're writing an HLA program, you can generally assume that people reading the source files you write are reasonably familiar with HLA's built-in types. By creating aliases of those type names, you make it harder for people who already know HLA to read and understand your programs because they have to mentally translate your new types to the more familiar type names. It might seem "cool" to use C++ type names or type names from some other programming language, but other people reading your programs might not share your enthusiasm for the renamed types.

One place where type isomorphisms might make sense is when you're creating a new type that is intended to be a subset of the full type (whose range you check at run time). For example, suppose you use a set of integers that must be in the range 0..31 for certain sections of your program. You could create a type definition like the following to let people know that variables of the specified type are supposed to lie in the range 0..31:

```
type
    smallInt_t :int8;  // Holds values in the range 0..31
```

At run time you could use the bound instruction to verify that values you assign to a *smallInt_t* object are actually in the range 0..31:

```
    bound( eax, 0, 31 );// Raises an ex.bound exception if not in the
range 0..31.
    mov( al, smallIntVar );
```

This example also demonstrates another common HLA programming convention: using an "_t" suffix on user-defined type identifiers.

10.2.4.3 newTypeID : typeID [list_of_array_bounds];

This form creates an array type with the specified number of elements. The *list_of_array_bounds* item is a list of one or more unsigned integer values that are greater than zero (and, generally, greater than one). If there are two or more array bound values, the type is a multi-

dimensional array type and the number of elements is the product of all of the array bound values. Note that HLA arrays are always indexed from zero to the array's bound value *minus one*. So an array declared as

10.2.4.4 newTypeID : procedure (<<optional_parameter_list>>);

A complete discussion of procedure pointer types appears in the chapter on procedures. Please see that document for a discussion of procedure pointer types. Like all pointer types, objects that are procedure pointers will consume four bytes in memory (and those four bytes typically hold the address of some procedure).

10.2.4.5 newTypeID : record <<record_field_declarations>> endrecord;

Please see the discussion of Record Data Types earlier in this document for examples of **record** type declarations, their syntax, and their use.

10.2.4.6 newTypeID : union <<union_field_declarations>> endunion;

Please see the discussion of Union Data Types earlier in this document for examples of **union** type declarations, their syntax, and their use.

10.2.4.7 newTypeID : class <<class_field_declarations>> endclass;

A complete discussion of **class** types appears in the chapter on Classes and Object-Oriented Programming in HLA. Please see that document for a discussion of class types.

10.2.4.8 newTypeID : pointer to typeID;

Please see the discussion of Pointer Data Types earlier in this document for examples of **pointer** type declarations, their syntax, and their use.

10.2.4.9 newTypeID : enum{ <<list_of_enumeration_identifiers>> };

Please see the discussion of Enumerated Data Types earlier in this document for examples of **enum** type declarations, their syntax, and their use.

10.2.5 The HLA VAR Declaration Section

The **var** section is where you declare automatic variables in an HLA **procedure, method, iterator,** or **program**. The HLA **var** section may not appear in the declaration section of an HLA **unit** or **namespace**. A **var** section may also appear in an HLA class, but the storage mechanism for class **var** objects is not the same as for procedures, methods, and iterators. Please see the chapter on Object-Oriented Programming for more details about **var** declarations in a class. The basic syntax for an HLA **var** section is the following:

```
var
    << variable declarations >>
```

or

```
var
   << variable declarations >>
endvar;
```

The syntax for the variable declarations is similar to type declarations; each variable declaration takes one of the following forms:

```
varID : typeID;
varID : typeID [ list_of_array_dimensions ];
varID : procedure (<<optional_parameter_list>>);
varID : record  <<record_field_declarations>> endrecord;
varID : union  <<union_field_declarations>> endunion;
varID : pointer to typeID;
varID : enum{ <<list_of_enumeration_identifiers>> };
```

These statements will allocate sufficient storage on the stack for each variable (*varID*) that you declare in the **var** section.

The HLA **var** section also supports an **align** directive; the syntax for the **align** directive in a **var** section is the following:

```
align( constant_expression );
```

The *constant_expression* must be a fully-defined constant expression that evaluates to a power of two that lies in the range 1..16. Technically, the only value that makes sense for the align expression is 4, as you will soon see.

The **var** section declares variables for which the HLA run-time code automatically allocates storage upon entry into a procedure (note: the HLA run-time system automatically allocates storage only if the **@frame** procedure option is enabled; otherwise it is the programmer's responsibility to actually allocate the storage). Automatic variable storage allocation is accomplished using a *standard entry sequence* into a procedure, such as the following

```
push( ebp );// Save old frame pointer
mov( esp, ebp );// Put new frame pointer value into EBP
sub( _vars_, esp );// Allocate storage for var variables on stack
```

Automatic variables (**var** objects, or *auto* variables) are referenced using negative offsets from the EBP (base pointer) register into the procedure's *stack frame* (also known as an *activation record*). The previous frame pointer (EBP) value is found at [EBP+0] and the auto variables are found on the stack below this location. Each variable is allocated some amount of storage (determined by the variables type) and the offset of the variable (from EBP) is computed by subtracting the variable's size from the offset of the previous object in the **var** section. Consider the following **var** declaration section; the comments tell you the offset to each of the objects from the EBP register (these offsets assume that there is no display):

```
var
   d  :dword;// offset = ebp-4
   s  :string;// offset = ebp-8
   u  :uns32;// offset = ebp-12
   i  :int32;// offset = ebp-16
   w  :word;    // offset = ebp-18
   b  :byte;    // offset = ebp-19
   c  :char;    // offset = ebp-20
```

The offset of each object is computed by subtracting the size of the object from the offset of the previous object in the **var** declaration section (the first object's offset is computed by subtracting the object size from zero, which is the offset of the saved EPB value).

Because the x86 supports a 1-byte offset (+/- 128 bytes) form of the "[EBP+offset]" addressing mode, your code will be slightly shorter if you group all your small variable objects at the beginning of the **var** declaration section and putting all your structured data types (e.g., arrays, records, unions, and character sets) near the end of **var** section. Consider the following two variable declaration sections:

```
var
    d   :dword;    // offset = ebp-4
    s   :byte[256];// offset = ebp-260
```

versus

```
var
    s   :byte[256];// offset = ebp-256
    d   :dword;    // offset = ebp-260
```

The instruction "mov(d, eax);" is three bytes shorter if you use the first set of declarations above (where *d's* offset is -4) because HLA can encode the offset in one byte instead of a double word. By putting the declarations of the smaller objects at the beginning of the **var** section, you can increase the number of variables that you can reference with a 1-byte displacement.

As HLA processes each variable in a **var** section, it computes the offset of that variable by subtracting the variable's size from the offset of the previous variable. If you mix different sized variables in the **var** section, you may not get optimal addresses for each of the variables. Consider the following **var** section and the corresponding variable offsets:

```
var
    b   :byte;     // offset = ebp-1
    w   :word;     // offset = ebp-3
    d   :dword;// offset = ebp-5
    q   :qword;// offset = ebp-13
```

Assuming that EBP points at an address that is a multiple of four (and it usually will), all of these variables will be misaligned except for **b**, the byte variable. One solution to this problem is to use the **align** directive to align each variable at an offset that is a multiple of that variable's size:

```
var
    b   :byte;     // offset = ebp-1
    align(2);
    w   :word;     // offset = ebp-4
    align(4);
    d   :dword;// offset = ebp-8
    align(8);
    q   :qword;// offset = ebp-16
```

Unfortunately, sticking an align directive before each variable declaration is a pain. Fortunately, the **var** declaration supports the same alignment options as **record** declaration. Consider the following:

```
var[4];
    b   :byte;     // offset = ebp-4
    w   :word;     // offset = ebp-8
    d   :dword;// offset = ebp-12
    q   :qword;// offset = ebp-24
```

Of course, allocating four bytes for each automatic variable can be wasteful; you can also do the following:

```
var[4:1];
```

```
    b   :byte;    // offset = ebp-1
    w   :word;    // offset = ebp-4
    d   :dword;// offset = ebp-8
    q   :qword;// offset = ebp-16
```

See the discussion of record type declarations earlier in the chapter on HLA Language Elements for more details about the alignment options.

There is one big issue concerning the use of the **var** section **align** statement and the alignment options: generally you may only assume that the stack pointer is aligned to a 4-byte address upon entry into a subroutine. Therefore, alignment values other than 1, 2, or 4 may not achieve the desired memory alignment for your automatic variables. If you absolutely must have your automatic variables aligned on a boundary greater than four, you will have to explicitly guarantee that the variable is properly aligned in the activation record. There are a couple of different ways to do this.

The first way is to allocate storage on the stack (using talloc), create an address into that storage area that you've aligned to the desired boundary, and then save a pointer to that storage in another automatic variable. For example, to create a 16-byte aligned object (e.g., for SSE objects that require 16-byte alignment), you could do the following:

```
var
    ptrToAligned16:pointer to lword;
    .
    .
    .
sub( 16, esp );
and( $ffff_fff0, esp );
mov( esp, ptrToAligned16 );
```

The "and($ffff_fff0, esp);" instruction ensures that ESP is situated at an address that is aligned on a 16-byte boundary. Note that this instruction might actually allocate up to 15 additional bytes (or, if the stack is aligned properly to begin with, up to 12 additional bytes) in order to guarantee that the new address is aligned to a 16-byte boundary.

Whenever using this technique to allocate storage for an aligned object, you must allocate the storage before pushing any other data you need to retrieve onto the stack. Because you don't really know how much storage these three instructions will actually allocate on the stack, you won't know where any data might be that you've previously pushed onto the stack. As such, you wouldn't be able to pop that data later on. The best way to avoid this problem is to allocate aligned data immediately upon entry into a procedure, iterator, or method, before any other stack operations take place:

```
procedure hasAlignedStorage; @nostackalign;
var
    ptrToAligned16:pointer to lword;
begin hasAlignedStorage;

    sub( 16, esp );
and( $ffff_fff0, esp );
mov( esp, ptrToAligned16 );
    .
    .
    .

end hasAligned16;
```

Note the use of the @nostackalign option. If you're created aligned data on the stack, you're already aligning ESP to an address that is a multiple of four. Therefore, there is no need for HLA to emit this instruction for you.

If you are creating your own stack frame upon entry into the procedure (e.g., you're using the @noframe option), then you should allocate storage for your aligned objects after you've created the stack frame/activation record:

```
procedure hasAlignedStorage; @noframe;
var
   ptrToAligned16:pointer to lword;
begin hasAlignedStorage;

   push( ebp );
   mov( esp, ebp );
   sub( _vars_, esp );
   sub( 16, esp );
and( $ffff_fff0, esp );
mov( esp, ptrToAligned16 );
   .
   .
   .

end hasAligned16;
```

Although this scheme allows you to allocate storage that is aligned on a 16-byte boundary (any other boundary that is a power of two is easy to achieve by modifying the **sub** and **and** instructions), this scheme isn't actually aligning the variables in the **var** section to a specific boundary. If you need to align the automatic variables themselves, it's going to take a bit more work to achieve. Achieving this goal requires that the stack be aligned to a given boundary *before* you call the subroutine. Unfortunately, you cannot simply align the stack pointer immediately upon entry into a subroutine, prior to building the activation record, because any parameters that the caller has pushed onto the stack must be accessible at fixed positions from EBP. If you align the stack upon entry into the code, you'll mess up the offsets to the parameters from EBP, thereby changing the assumptions HLA makes about where those parameters' values lie. Therefore, you must align the stack pointer to a desired address before pushing any parameters onto the stack. This means that the calling code will be responsible for aligning the stack and this has to be done on *each* call to the subroutine.

The task is to set up the stack pointer so that when it pushes EBP on the stack (while setting up the activation record) the address of the old EBP value on the stack is a multiple of whatever alignment you need. Unfortunately, you cannot simply align the stack pointer before the call because the subroutine's parameters, return address, and the EBP value itself consume space on the stack that may cause the alignment to change. Therefore, you will need to adjust to the stack pointer prior to the call so that ESP is aligned to an appropriate address after the caller has pushed the parameters, return address, and EBP has been preserved on the stack. For example, consider the following HLA procedure:

```
procedure p(i:int32);
var[16];
   b:byte;
   w:word;
   d:dword;
   l:lword;
begin p;
   .
   .
   .
end p;
```

The goal here is to align each of the automatic variables to an address that is a multiple of 16 bytes. Upon entry into the body of the procedure, there will be 12 bytes pushed onto the stack - four bytes for parameter *i*, four bytes for the return address, and four bytes with the old EBP value. Therefore, simply aligning ESP to some multiple of four before the call will not work because when the call to the procedure occurs, an additional 12 bytes wind up on the stack, leaving ESP misaligned. What has to be done is to align ESP to a multiple of 16 bytes and then drop the stack pointer down four bytes so that when the calling sequence pushes those 12 bytes onto the stack, ESP winds up properly aligned on a 16-byte boundary. This can be done with the following code sequence (that calls procedure *p*):

```
and( $FFFF_FFF0, esp );// Align ESP to 16-byte boundary
sub( 4, esp );             // 4 + 12 bytes keeps it 16-byte aligned
p( 2 );                          // Call p.
```

Alas, "4" is a magic number here that probably won't make much sense to the reader of this code. Furthermore, if you ever change the number or types of *p's* parameters, "4" might no longer be the correct value to use here. Fortunately, HLA's compile-time language provides a compile-time function, **@parms**, that returns the number of parameter bytes for the procedure whose name you specify as an argument. So we can use the following generic version to properly align the stack on a 16-byte boundary:

```
and( $FFFF_FFF0, esp );
sub( 16-((@parms(p)+8) & $F), esp );
p( 2 );
```

The "@parms(p)+8" portion of the expression is the total number of bytes pushed on the stack up to the point where EBP will be pointing in the activation record. The "(@parms(p)+8) & $F" computes this value modulo 16 because we never need to push more than 15 bytes in order to align ESP to a 16-byte boundary. Finally, "16-((@parms(p)+8) & $F)" computes the number of bytes we must drop the stack down in order to guarantee 16-byte alignment upon entry into the subroutine.

We could make one additional improvement to this code. On occasion, the expression "16-((@parms(p)+8) & $F)" will evaluate to zero and there is no reason at all to execute the **sub** instruction. Because this is a constant expression, we can determine that it is zero at compile time and use conditional assembly to eliminate the **sub** instruction:

```
and( $FFFF_FFF0, esp );
#if( (16-((@parms(p)+8) & $F)) <> 0 )

    sub( 16-((@parms(p)+8) & $F), esp );

#endif
p( 2 );
```

Remember, you have to execute this instruction sequence before each call to procedure *p* in order to guarantee that *p's* local variables are properly aligned on a 16-byte boundary. As it's easy to forget to execute this sequence prior to calling *p*, you might want to consider writing a macro to invoke that will automatically do this for you. Consider the following code:

```
#macro p( _i_ );
and( $FFFF_FFF0, esp );
#if( (16-((@parms(_p)+8) & $F)) <> 0 )

    sub( 16-((@parms(_p)+8) & $F), esp );

#endif
p( _i_ );
#endmacro
```

```
procedure _p(i:int32);
var[16];
    b:byte;
    w:word;
    d:dword;
    l:lword;
begin _p;
    .
    .
    .
end _p;
    .
    .
    .
p(2);
```

One problem with aligning the stack in this manner is that the code suffers from "stack creep". Each time you call procedure *p* it might drop the stack down as many as 12 bytes. If this isn't a problem (e.g., if you call *p* from within some other procedure that cleans up the stack upon returning to its caller) then you can ignore the stack creep. However, if you've pushed data onto the stack that you need to pop after the call to p, or if you're calling p within a loop and that would cause considerable stack creep, then you'll want to save ESP's value in a local (automatic) variable in the calling code and restore ESP upon return, e.g.,:

```
mov( esp, espSave );
p(2);               // This is the macro from above!
mov( espSave, esp );
```

Be sure to use a local automatic, not a static, variable for *espSave*. Also, avoid the temptation to use **push** and **pop** to preserve ESP's value, remember that ESP is modified by the call to *p* and you won't be popping what you've pushed.

One last feature available in the **var** section is the ability to set the starting offset of the activation record. By default, HLA uses the offset zero as the base offset of the activation record. HLA assigns local (automatic) variables negative offsets from this base offset and parameters positive offsets from the base offset. Using the following syntax, you can change the base offset from zero to any other signed integer value you choose:

```
var:= <<signed integer expression>>;
    << var declaration section>>
```

The first local variable you declare in the << var declaration section >> will have the offset you specify by *<<signed integer expression>>*. *Note that HLA will not first subtract the size of the first object from your base offset as it normally does for the automatic variables you declare*. It uses the value you supply as the offset of the first variable you declare. Also note that this syntax does not change the offsets assigned to the parameters for the procedure. Therefore, EBP must point at the same location (at the old value of EBP immediately below the return address) it would if you didn't set the starting offset.

In general, this syntax is far more useful for **record** data structures than it is for **var** activation records, but it can be useful if you want to explicitly declare the saved EBP value and the display (if one is present). For example:

```
procedure p(i:int32);
var := 0;
    saveEBP:dword;
    display:dword[2];
    b         :byte;
```

```
        w           :word;
        d           :dword;
        l           :lword;
    begin _p;
        .
        .
        .
    end _p;
```

The var declaration section also supports the following (rarely-used) **@nostorage** syntax:

```
varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;
```

These declarations assign the current offset (after subtracting the size of the object) into the activation record to the variable you've declared, but they do not reserve any storage for such variables. As a result, variable declarations with the **@nostorage** option in the **var** section overlay the following variable declaration(s). Consider the following variable declarations:

```
    var
        w:word;  @nostorage;
        b:byte;  @nostorage;
        d:dword;
```

The *b* variable will be sitting at offset -1, the *w* variable will be at offset -2, and the *d* variable will be sitting at offset -4. Note that these variables overlap one another in memory. Be very careful when using the **@nostorage** option in the **var** declaration section. If you declare a large object using the **@nostorage** option and you don't declare sufficient storage in variables after that object, accessing that object may wind up wiping data in "no man's land" on the stack.

Note that because offsets into the activation record are negative, the **@nostorage** option behaves differently in the **var** section from the way it works in the **static, storage,** and **readonly** sections. If you have a byte variable with an **@nostorage** option followed by a dword variable, the byte variable will be sitting in the H.O. byte of the dword object (rather than in the L.O. byte position, as it would in a **static, storage,** or **readonly** section). For this reason, you'll rarely see the **@nostorage** option used in a **var** section.

10.2.6 The HLA STATIC Declaration Section

The **static** section is where you declare static/data variables in an HLA **namespace, class, procedure, method, iterator,** or **program**. The basic syntax for an HLA **static** section is the following:

```
static
    << static variable declarations >>
```

or

```
static
    << static variable declarations >>
endstatic;
```

Each static variable declaration can take one of the following forms:

```
Uninitialized forms:
```

```
varID : typeID;
varID : typeID [ list_of_array_dimensions ];
varID : procedure (<<optional_parameter_list>>);
varID : record  <<record_field_declarations>> endrecord;
varID : union  <<union_field_declarations>> endunion;
varID : pointer to typeID;
varID : enum{ <<list_of_enumeration_identifiers>> };


Initialized forms:
varID : typeID := <<constant expression>>;
varID : typeID [ list_of_array_dimensions ] := <<constant expression>>;
varID : procedure (<<optional_parameter_list>>) := <<constant
expression>>;
varID : pointer to typeID := <<constant expression>>;
varID : enum{ <<list_of_enumeration_identifiers>> }  := <<constant
expression>>;


No allocation forms:
varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;
varID : enum{ <<list_of_enumeration_identifiers>> }; @nostorage;


External forms:
varID : typeID; external;
varID : typeID; external( "external_name" );
varID : typeID [ list_of_array_dimensions ]; external;
varID : typeID [ list_of_array_dimensions ]; external( "external_name" );
varID : procedure (<<optional_parameter_list>>); external;
varID : procedure (<<optional_parameter_list>>); external(
"external_name" );
varID : pointer to typeID; external;
varID : pointer to typeID; external( "external_name" );
```

The first set of declarations above creates non-initialized static variables. "Non-initialized" means that the program does not explicitly initialize these static variables before the program begins execution; in fact, the system initializes all non-initialized static objects to zero (or all zero bits) when the program loads into memory. Although you can safely assume that all non-initialized static variables contain zero bits, it's still wise to explicitly initialize static variables you expect to contain zero; leave the non-initialized forms of the static variable declaration for those variables whose initial value is completely irrelevant (e.g., because the program will initialize the value before ever using it).

Here are some examples of non-initialized variable declarations:

```
static
    i         :int32;
    user   :someUserType;
    ary    :char[ 3 ];
    usrAry :someUserType[2];
    procPtr:procedure (cnt:uns32);
    quickRec:record
                    a   :char;
```

```
                    b   :boolean;
                    c   :char;
                endrecord;
    quickUn:union
                    a   :char;
                    b   :boolean;
                    c   :char;
                endunion;
    charPtr:pointer to char;
    usrPtr :pointer to someUserType;
    colors :enum{ red, green, blue };
```

Static variables also support initialization via some constant expression using the syntax from the second group of declarations given earlier. The type of the constant expression must be compatible with the type of the static variable you are declaring (that is, the type must match or HLA must be able to convert the constant's type to the specified type at compile time). Here are some examples:

```
type
    someUserType :record
                        b:boolean;
                        c:char;
                        w:word;
                        d:dword;
                    endrecord;

procedure p( parameter:uns32 );
begin p;
    .
    .
    .
end p;

static
    i         :int32 := -4;
    user   :someUserType := someUserType:[ false, 'a', 0, 1];
    ary    :char[ 3 ] := ['a', 'b', 'c'];
    usrAry :someUserType[2] :=
                [
                    someUserType:[ false, 'a', 0, 1],
                    someUserType:[ true, 'b', 1, 0]
                ];

    procPtr:procedure (cnt:uns32) := &p;
    charPtr:pointer to char := &ary[0];
    usrPtr :pointer to someUserType := &user;
```

You will notice that you cannot provide initializer constants for all of the static variable declarations. In particular, you cannot assign a constant to a static variable directly declared as a **record**, **union**, or **enum** object. However, as this last example demonstrates, this isn't a limitation because you can easily create a user-defined type that is a **record, union,** or **enum** type and use that type in a static variable declaration (e.g., as was done with *someUserType* in this example).

The third syntactical form (the "no allocation" forms) create a typed label in memory without explicitly allocating storage for that static variable. As a result, a variable with the **@nostorage** option will be sitting at the same memory location as the following variable(s) you declare in memory. Consider the following example:

```
static
```

```
    b           :byte; @nostorage;
    u           :uns32;@nostorage;
    i           :int32;
```

The *b, u,* and *i* variables will all be sitting at the same starting address in memory; any modification to one of these variables will modify the others, as well. This declaration is almost equivalent to putting these three variables into a **union** data type.

The static declaration section also allows the declaration of external (and public) variables (see the "External Forms" syntax given earlier). Each external declaration can take one of two forms, one using the "external;" declaration and one using the "external("*external_name*");" declaration. For example:

```
static
    b           :byte; external;
    u           :uns32;external( "u_var" );
```

When **external** appears by itself, HLA will use the declared variable's name (e.g., *b* in this example) as the external name. Whenever you use the second form (with the string argument), HLA will use the declared name within the current source file and use the string name supplied as the external argument as the external name (e.g., *u_var* in place of *u* in this example).

When you compile and link your program, the system assumes that you have declared all external static variables in some other object module (that you link with the file containing the external declaration). The file containing the actual variable declaration must define the symbol as a *public* symbol. You create a public symbol by having both an external definition of the symbol and a regular declaration of that symbol, e.g.,

```
static
    b           :byte; external;
    b           :byte := 1;

    u           :uns32;external( "u_var" );
    u           :uns32 := 2;
```

All public and non-external variables in a **static** section will consume the corresponding amount of space in the executable program's disk file. This is true even if you don't explicitly assign a value to a **static** object using the initializer syntax. If you don't explicitly assign a value to a **static** variable, HLA will write a zero to the corresponding location on the disk. This is how the system initializes those variables when the program begins running: it simply copies the data from the disk file to the location in memory where the variable will be accessed. As non-initialized variables have zeros written to the disk file at their corresponding locations, the operating system will load those zeros into the memory locations reserved for such variables, thus initializing them to zero.

An HLA **static** declaration section can also appear in the body of a procedure or program. In such a case, you must explicitly terminate the static declaration section with an **endstatic** clause. Here's an example of such a declaration section:

```
procedure p;
begin p;

    .
    .
    .
    static
        b           :byte := 1;
        w           :word;
        d           :dword; external;
```

```
    endstatic;
    .
    .
    .
end p;
```

As far as HLA is concerned, such declarations are treated as though you'd place them in the declaration section of that procedure (except you cannot access the variables until after they are declared). The main reason for allowing this type of static declaration section is to support variable declarations in macros that might need to declare static variables but are invoked within the body of the procedure. It would be unusual for you to explicitly declare static variables this way.

10.2.7 The HLA STORAGE Declaration Section

The **storage** section is where you declare uninitialized static variables in an HLA **namespace, class, procedure, method, iterator,** or **program**. The basic syntax for an HLA **storage** section is similar to that for **static** except you are not allowed to initialize any variables you declare in the **storage** section. The syntax is the following:

```
storage
    << storage variable declarations >>
```

or

```
storage
    << storage variable declarations >>
endstorage;
```

Each **storage** variable declaration can take one of the following forms:

```
Standard forms:
varID : typeID;
varID : typeID [ list_of_array_dimensions ];
varID : procedure (<<optional_parameter_list>>);
varID : record  <<record_field_declarations>> endrecord;
varID : union  <<union_field_declarations>> endunion;
varID : pointer to typeID;
varID : enum{ <<list_of_enumeration_identifiers>> };


No allocation forms:
varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;
varID : enum{ <<list_of_enumeration_identifiers>> }; @nostorage;


External forms:
varID : typeID; external;
varID : typeID; external( "external_name" );
varID : typeID [ list_of_array_dimensions ]; external;
varID : typeID [ list_of_array_dimensions ]; external( "external_name" );
varID : procedure (<<optional_parameter_list>>); external;
```

```
varID : procedure (<<optional_parameter_list>>); external(
"external_name" );
varID : pointer to typeID; external;
varID : pointer to typeID; external( "external_name" );
```

Other than you cannot assign an initial value to a **storage** variable, the declaration of the variables in the **storage** section is identical to that of the **static** section. Please see the discussion in the **static** section for more details.

As far as your program is concerned, **storage** variables are static objects exactly like variables you declare in a **static** section. The only real difference between variables you declare in a **storage** section and those you declare in a **static** section is that the disk file holding the program's data and code does not contain any data for the individual variables. Instead, HLA makes note of the number of bytes for all your **storage** variable declarations and stores this size in the object file it produces. When the operating system loads your program into memory, it makes note of this size and allocates a sufficient amount of space for these "BSS" (Block Started by a Symbol - an ancient assembly language term) variables and then writes zeros to that block of storage so that the variables are all initialized to zero when the program begins running. Although your program will take the same amount of storage in memory regardless of whether you declare your variables in the **storage** or **static** section, you may save some disk space in the executable file if you declare your uninitialized variables in the **storage** section rather than the **static** section.

An HLA **storage** declaration section can also appear in the body of a procedure or program. In such a case, you must explicitly terminate the static declaration section with an **endstorage** clause. Here's an example of such a declaration section:

```
procedure p;
begin p;

    .
    .
    .
    storage
       b         :byte;
       w         :word;
       d         :dword; external;
    endstorage;
    .
    .
    .
end p;
```

As far as HLA is concerned, such declarations are treated as though you'd place them in the declaration section of that procedure (except you cannot access the variables until after they are declared). The main reason for allowing this type of storage declaration section is to support variable declarations in macros that might need to declare storage variables but are invoked within the body of the procedure. It would be unusual for you to explicitly declare storage variables this way.

10.2.8 The HLA READONLY Declaration Section

The **readonly** section is where you declare static read-only values in an HLA **namespace, class, procedure, method, iterator,** or **program**. The basic syntax for an HLA **readonly** section is the following:

```
readonly
```

```
    << readonly variable declarations >>
```

or

```
readonly
    << static variable declarations >>
endreadonly;
```

Each **readonly** object declaration can take one of the following forms:

```
Initialized forms:
varID : typeID := <<constant expression>>;
varID : typeID [ list_of_array_dimensions ] := <<constant expression>>;
varID : procedure (<<optional_parameter_list>>) := <<constant
expression>>;
varID : pointer to typeID := <<constant expression>>;
varID : enum{ <<list_of_enumeration_identifiers>> }  := <<constant
expression>>;


No allocation forms:
varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;
varID : enum{ <<list_of_enumeration_identifiers>> }; @nostorage;


External forms:
varID : typeID; external;
varID : typeID; external( "external_name" );
varID : typeID [ list_of_array_dimensions ]; external;
varID : typeID [ list_of_array_dimensions ]; external( "external_name" );
varID : procedure (<<optional_parameter_list>>); external;
varID : procedure (<<optional_parameter_list>>); external(
"external_name" );
varID : pointer to typeID; external;
varID : pointer to typeID; external( "external_name" );
```

Note that there are no non-initialized forms that have storage allocated for them. A **readonly** object must have an initializer attached to it, have the **@nostorage** attribute (in which case it inherits the initial value of the following **readonly** object you declare), or it must be an external declaration.

HLA places all objects you declare in a **readonly** section into memory that the operating system write protects. Any attempt to store data into a **readonly** object at run time will result in a segmentation/access violation fault. This is enforced by the operating system, not by HLA. You can create an instruction that will store data into a **readonly** object and HLA will compile the program just fine. When you try to run the program, however, it will generate an exception when you attempt to execute that instruction.

An HLA **readonly** declaration section can also appear in the body of a procedure or program. In such a case, you must explicitly terminate the **readonly** declaration section with an **endreadonly** clause. Here's an example of such a declaration section:

```
procedure p;
begin p;
```

```
    .
    .
    .
    readonly
       b          :byte := 1;
       w          :word := 2;
       d          :dword; external;
    endreadonly;
    .
    .
    .
end p;
```

As far as HLA is concerned, such declarations are treated as though you'd place them in the declaration section of that procedure (except you cannot access the variables until after they are declared). The main reason for allowing this type of **readonly** declaration section is to support variable declarations in macros that might need to declare **readonly** variables but are invoked within the body of the procedure. It would be unusual for you to explicitly declare **readonly** variables this way.

10.2.9 The HLA PROC Declaration Section

The **proc** section is where you declare "new style" procedures, iterators, and methods. The chapter on procedures goes into detail about the proc section, please see the discussion of the **proc** section in that chapter.

10.2.10 THE HLA NAMESPACE Declaration Section

HLA supports a special declaration section known as a **namespace**. A namespace is a collection of declarations that HLA gathers together under a single identifier (the **namespace** identifier). A namespace declaration uses the following syntax:

```
namespace userNamespaceID;

    << namespace declarations >>

end userNamespaceID;
```

userNamespaceID is an identifier you associate with the namespace declarations; you will use this identifier when referencing members of the namespace in your application. The body of the namespace, << *namespace declarations* >>, can be any of the following declaration sections:

const

val

type

static

readonly

storage

proc

old-style procedure, method, and iterator declarations

Note that **label** and **var** declaration sections are illegal in a namespace. In addition, you cannot nest **namespace** declarations (that is, **namespace** declarations are not legal in a **namespace**).

Namespace declarations should always appear an lex-level one in a **program** or **unit**. In HLA v2.x namespaces have a relatively kludged implementation and strange things might happen if you

declare namespaces within classes, procedures, methods, or iterators; namespace declarations have not been extensively tested in such cases and will probably fail to work properly.

An unusual feature about namespaces is that the **namespace** identifier does not have to be unique within its scope (that is, at lex level one). You can have multiple **namespace** declarations in a program with the same **namespace** identifier. HLA will simply combine these separate namespaces into a single unit. This is useful, for example, when you've got several different include files and each include file contains a common **namespace** declaration with the intent of constructing one big name space from the three separate ones. Although the **namespace** identifier need not be unique, all the declarations in a **namespace** with a given identifier must be unique. That is, you cannot declare two objects with the same name in a single name space.

One of the principle purposes of an HLA **namespace** is to prevent *name space pollution*. As your applications increase in size, and especially as you start to link in libraries of subroutines you (or other people) have created, it becomes difficult to avoid reusing names that other code is already using. For example, you might want to write a *put* macro or procedure to output data in some special way. However, *put* is a very common name (for example, the HLA Standard Library uses it) so you'd probably have to dream up a different name if you wanted to use this identifier. This is where name spaces come to the rescue. You can encapsulate every instance of the *put* identifier in a separate namespace and avoid the conflicts. For example, the HLA Standard Library uses the *put* identifier all over the place, but it's buried in the *stdout, stderr, fileio, str,* and other name spaces, so these identifiers don't conflict with one another.

To access an identifier that is a member of a namespace, you use the same *dot notation* that HLA uses for **record, union**, and **class** field access. To access a field from a **namespace** you specify the name space identifier, a period (dot), followed by the field name. For example, to invoke the *put* macro in the HLA Standard Library *stdout* namespace, you use the (very familiar) sequence *stdout.put*. If you create your own **namespace**, you simply substitute your name space identifier and the field name, e.g.,:

```
program nsDemo;

namespace myNamespace;

    static
       x:dword;

    procedure pp( p:dword );
    begin pp;
    end pp;

end myNamespace;

begin nsDemo;

    myNamespace.pp( myNamespace.x );

end nsDemo;
```

As noted above, namespaces in HLA have a somewhat kludged implementation. One artifact of this implementation is that within a **namespace** no global symbols (symbols declared outside the **namespace**) are directly visible. This includes some HLA-defined symbols (such as **true** and **false**) in addition to any symbols you've defined. If you need to reference any symbols defined outside the namespace within code (or expressions) inside the namespace, you will need to prepend the **@global:** string to the global symbol; otherwise, HLA will generate an *unknown symbol* error. Here is an example of using the **@global** modifier:

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

namespace myNamespace;
```

```
    static
       b:boolean := @global:true;

    procedure pp( p:@global:array );
    begin pp;
    end pp;

end myNamespace;

static
    a  :array;

begin nsDemo;

    myNamespace.pp( a );

end nsDemo;
```

A **namespace** declaration section may contain external- and forward-declared objects. Forward and public objects must be defined somewhere in the namespace within the current compilation, but you could have the external/forward definition in one instance of a particular namespace and the actual declaration of the object in another instance of that same namespace, e.g.,

```
program nsDemo;
namespace myNamespace;

    static
       b:boolean; external;

    procedure pp( p:dword ); forward;

end myNamespace;

// Assume some other code is here...

namespace myNamespace;

    static
       b:boolean;

    procedure pp( p:dword );
    begin pp;
    end pp;

end myNamespace;

begin nsDemo;
end nsDemo;
```

This example is rather trivial, but it's not hard to imagine a better one. The HLA Standard Library include files, for example, contain dozens of **namespace** declarations containing external entries. The actual source code for the HLA Standard Library contains the actual implementation within a **namespace** declaration section (in a **unit**).

One big advantage to using namespaces is that they improve HLA's compilation speed when dealing with a large number of symbols. Namespaces use a special symbol table lookup algorithm that is much faster that the standard symbol table lookup algorithms that HLA uses for symbols

defined outside a namespace. Using namespaces to encapsulate a large number of symbols can dramatically improve compile times. For example, the w.hhf header file (that encapsulates all of its identifiers in the *w* namespace) used to take about 45 seconds to process on a Pentium IV processor, prior to putting all the symbols into a namespace. After adding namespaces to HLA, the compile time was reduced to a couple of seconds. So if you're creating a large project with hundreds or thousands of data variables and other symbols, you might want to consider sticking those symbols into a namespace in order to reduce compilation time.