HLA v2.0 Symbol Table Design Documentation

This document describes the internal operation of HLA v2.0's symbol table routines and data structures. This document assumes that the reader is familiar with compiler theory and terminology. This document also assumes that the reader is comfortable with the HLA language and 80x86 assembly language in general.

The symbol table module is responsible for maintaining a database of program identifiers during the compilation of an HLA program. The symbol table data structure and the symbol table functions faithfully implement the semantics required by the HLA language. This document will explain those semantics, describe the symbol table data structures, and also describe the algorithms needed to implement those semantics. The reader should have read the accompanying documentation on the HLA lexical analyzer (lexer) prior to reading this documentation.

Identifier Semantics in an HLA Program

Symbols in an HLA program exhibit certain semantics that you must understand in order to make sense of the symbol table algorithms and data structures. This section will carefully described those semantics so that you can understand the rest of this document.

Scope

HLA is a *block-structured language*. This means that we can lexically divide a set of HLA source modules that comprise a single program into several lexically independent blocks. This structure of an HLA program has a large bearing on the design and implementation of the HLA symbol table routines.

At the coarsest level of granularity, an HLA program is broken up into source files. Although an HLA program could consist of as little as a single source file during compilation, most HLA programs are actually made of multiple source modules (this is true even if the programmer believes the program contains a single source file; most HLA programs use the HLA Standard Library and the Standard Library routines themselves appear in different source modules). A typical HLA program consists of a single **PROGRAM** module and one or more **UNIT** modules[1]. Symbols appearing within a source module are always *local* to that module (that is, other modules cannot reference those symbols) unless the programmer explicitly declares those symbols to be external. Therefore, even at this coarsest level we see two different types of symbols: those that are usable only within a single source file module, and those that are usable within multiple modules.

We'll use the term *scope* to describe the visibility of symbols in an HLA source file. Typical symbols in a **PROGRAM** or **UNIT** have file scope; that is, the visibility (or usability) of these symbols is limited to the source file in which they are defined. It is possible, however, to define symbols with *external scope*. A module can reference a symbol with external scope even if that symbol's definition is in a different source file. In HLA, of course, you use the **EXTERNAL** attribute to explicitly tell the compiler that a symbol has external scope. If a symbol in a source module does not have the **EXTERNAL** attribute, then the symbol only has file scope.

Note that if a source file contains both an external declaration and a file scope declaration, then that symbol is *public* to that given source file. There must be exactly one public declaration for each external symbol in a set of HLA source modules that comprise a single program.

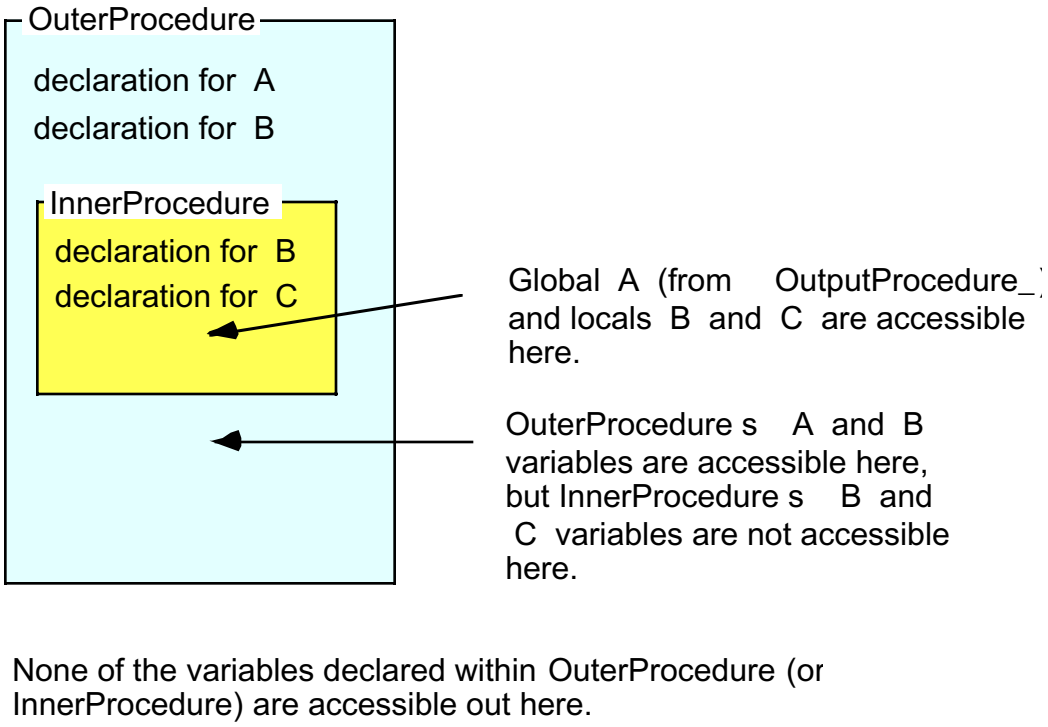
Within a source file, the scope of an identifier is limited in several ways. Symbols you declare within a namespace have *namespace scope*. You may only refer directly to such symbols within the namespace that contains them or by providing a "dot-path" to the symbol using the namespace identifier.

---

1. In theory, an HLA program does not require a PROGRAM module; it could consist of a set of UNITs linked together if the programmer understands the operating systems' run-time system. However, this would be an extremely rare situation and doesn't really change the discussion, so we will ignore this here. It is also possible to obtain a slightly different module topology if you consider linking HLA UNITs with code from other languages. We will not consider that situation here for the same reason - it doesn't affect this discussion in any major way.

Symbols that you declare within a program unit (procedure, iterator, or method) use "local scope." Variables with local scope are not visible outside the boundaries of the program unit in which you declare them. It is not an error for a local variable to have the same name as a global variable. Within the scope of the local variable (that is, within the program unit in which you define the local variable) the local name takes precedence over the global name. HLA does not provide a mechanism for overriding the scope of the local name in order to access the global name.

If you declare one program unit within another (e.g., you nest procedure declarations), then the local symbols in the outer program unit are global to the nested program unit declaration. As long as a global symbol is not redefined in the nested program unit, that symbol's name is visible within the nested program unit.



Note that external declarations are never allowed within a procedure, iterator, or method. External declarations are legal only at the global level of a PROGRAM or UNIT.

Record, union, and class declarations provide another name space that defines the scope of an identifier within a program. Identifiers within one of these data structures (i.e., the field names) are accessible via the "dot-path" syntax for record, union, and class objects. The names within a given record, union, or class must be unique to that particular data structure, but you can reuse names in different structures and outside the structure.

The scope of identifiers within a record, class, or union, are futher affected by the PRIVATE section of these data structures. Consider the following record declarations:

```
base: record
      public1:uns32;
      public2:uns32;
   private:
      private1:uns32;
      private2:uns32;
endrecord;

derived: record inherits( base )
      public3:uns32;
```

```
        private1:uns32;
    endrecord;
```

The PRIVATE reserved word tells HLA that the following fields in the record will not be visible in any records that inherit the fields of *base*. Therefore, records that inherit from *base* may freely reuse the identifiers *private1* and *private2* (as the derived record does above). Note that the derived records/classes/unions still reserved space for the private fields from the base data structure, however, the names are no longer accessible in the derived structure[1]

HLA's multi-part (context-free) macros introduce some interesting twists to the variable scope model. Consider, first, the following simple multi-part macro declaration:

```
macro multipart:x;

        << some text to expand >>

keyword keywrd:y;

        << some text to expand #2 >>

terminator termntr:z;

        << some text to expand #3 >>

endmacro;
```

Also consider the following invocation of the above macro:

```
multipart

        << body text, part 1 >>

  keywrd

        << body text, part 2 >>

termntr;
```

First, note that the symbols *keywrd* and *termntr* are not available until the user actually invokes the *multipart* macro. This is a form of dynamic scoping (dynamic with respect to HLA's compile-time language) insofar as the declarations of the symbols do not control the scope, but rather the usage of the symbols within the source file. Note that once the compiler encounters the *termntr* symbol in the sequence above, the *keywrd* and *termntr* symbols are no longer visible (or "in scope") to the program.

Macros also have their own local symbols. In the example above, *x, y,* and *z* are all examples of local macro symbols. The scope of a local macro symbol is somewhat complex and challenging to describe (not to mention, implement). Like the KEYWORD and TERMINATOR identifiers, the scope of a local macro symbol is dynamically defined and the behavior isn't entirely intuitive.

Local macro symbols you declare in a KEYWORD or TERMINATOR section are the easiest to understand because their behavior is very similar to local symbols within a procedure. Local symbols in these sections (*y* and *z* in the example above) are visible only within the macro text expansion section associated with the KEYWORD or TERMINATOR sections (<< some text to expand #2 >> and << some text to expand #3 >> in the example above). Note that these symbols are not available outside the macro declaration (and, in particular, the symbol y is not defined in the << body text, part 2 >> section).

---

1. HLA v1.x used the OVERRIDES keyword to achieve this result.

Local symbols in a MACRO declaration behave differently than local symbols in a KEYWORD (or TERMINATOR) section. Like the KEYWORD and TERMINATOR sections, local symbols you define with the MACRO statement are visible in the text expansion associated with the MACRO statement (<< some text to expand #1 >> in the example above); however, the local symbols you declare in a MACRO statement are also visible within all text expansion sections of a multi-part macro. That is, the identifier *x* in the example above is visible not only in << some text to expand #1 >>, but also << some text to expand #2 >> and << some text to expand #3 >>. You are not allowed to redefine local macro symbols within the same macro (e.g., you cannot redeclare *x* in the KEYWORD or TERMINATOR sections).

One of the more interesting aspects of local symbols you declare with the MACRO statement (in a multi-part macro) is that they are also visible between the initial macro invocation and the closing terminator symbol within your program's body. For example, the local macro symbol *x* is available within the << body text, part 1 >> and << body text, part 2 >> sections in the example above. If you've declared a symbol *x* in the current program unit, the macro's identifier overrides the scope of this symbol between the initial macro invocation (*multipart*) and the terminating symbol (*termntr*).

HLA's scoping rules have a big impact on the design of the symbol table data structures and algorithms. To understand the reason behind many of the compiler's design decisions, you should be comfortable with the semantics behind identifier visibility and scope in HLA. For more information, please consult the HLA documentation.

Generic Symbol Table Data Structures

The following paragraphs describe the data and meta-data we must keep for each symbol in the HLA symbol table:

next, left, right

These fields contain pointers to other symbol table entries. The *next* field implements a linear list of symbols within a particular symbol table. HLA uses the *left* and *right* fields to implement a binary search tree in the symbol table. Note that some algorithms (e.g., making a copy of a symbol table, dumping symbols, processing parameters, and symbol insertion) work best with a linear list of symbols; other algorithms (e.g., searching for a symbol) are much more efficiently implemented using a binary tree. Therefore, the HLA compiler imposes both data structures on the symbol tables it builds. Then a given operation can use whichever algorithm is more appropriate for the task at hand.

lcname, trueName

These two fields are (HLA) strings that specify the identifier associated with this symbol table entry. The *lcname* field contains the name in all lower case characters; the *trueName* field contains the symbol using the exact spelling (with respect to case) of the identifier in the source file. When searching for a symbol in the symbol table, HLA first converts the target string to lower case and then searches through the symbol table comparing the test string against the *lcname* field of the entries in the symbol table. If HLA finds a match, it then compares the original identifier against the string in the *trueName* field and reports a "case neutrality" violation if the second string comparison fails. This is how HLA enforces case neutrality of identifiers[1].

---

1. Case neutrality means that HLA uses a case sensitive comparison for identifiers so that all uses of an identifier within a source file must exactly match the original declaration. However, HLA does not treat identifiers whose only difference is alphabetic case as distinct; instead, HLA reports an error if you attempt to use such identifiers in your program.

symType, baseType, localSyms, inhType, equateLabel

These fields are all different names for the same physical pointer (i.e., they all belong to a union). This is a pointer to a symbol table entry and the meaning of the pointer depends upon the context, specifically the symbols classification.

For structures (records, classes, and unions) the *inhType* field contains a pointer to the base object (that is, a pointer to the object from which the current structure inherits fields). If the current structure symbol does not inherit any fields from some base structure, then this field will contain NULL.

For array types, the *baseType* field contains a pointer to the type entry for the array element's type. You may identify array types by looking at the *numElements* field of the symbol table data type. If *numElements* contains zero, then the object is scalar and the value of *baseType* is meaningless; however, if *numElements* is non-zero, then the *baseType* field points at the element type for this array. Note that HLA v2.x implements multidimensional arrays as "arrays of arrays." Therefore, the *baseType* field for a two-dimensional array object will point at the data type that specifies the second dimension of the array.

If the current symbol is a program unit identifier (procedure, method, iterator, program, or unit) or a NAMESPACE or a SEGMENT, then the *localSyms* field points at the local symbols for that program unit. Note that this field points at the local symbol table tree for that program unit (i.e., searching through the sub-symbol table will use a binary search).

For data objects that have some sort of type associated with them (i.e., scalar constants and variables), the *symType* field contains a pointer to the symbol table entry that defines the symbol's type. Note that the pType field will usually contain the equivalent type information as well, in a more compact form.

For LABEL objects, this field will contain a pointer to another label symbol table entry if the current label is equated to that other label. If the current symbol is not equated to another label, this field will contain NULL.

Certain objects (e.g., macros) do not have any associated symbol type information. For these objects, the *symType* field will contain NULL. For example, a procedure's symbol table entry doesn't have a data type associated with it, so the *symType* field contains NULL (though the *pType* field will contain *proc_pt* in this instance). The important thing to note is that the *symType* field doesn't necessarily contain a valid pointer; it may contain NULL.

seg

If the current object is a STATIC, READONLY, STORAGE, or SEGMENT variable, then this field points at the symbol table entry for the segment that contains this symbol. Note that STATIC objects belong to the *data* segment, READONLY objects belong to the *readonly* segment, and STORAGE objects belong to the *bss* segment (unless the programmer has changed the default names for these segments). For SEGMENT symbol table entries (including *data* (SEGMENT), *readonly* (READONLY), and *bss* (STORAGE) symbol table entries, HLA stores a pointer to the beginning of the linear list for the segment in this field. All other symbols store a NULL in this field.

segList, equateList

If the current object is a STATIC, READONLY, STORAGE, or SEGMENT variable, then the *segList* field points at the next variable in the linear list of symbols associated with this segment. For SEGMENT symbol table entries (including *data* (SEGMENT), *readonly* (READONLY), and *bss* (STORAGE) symbol table entries, HLA stores a pointer to the last entry of the linear list for the segment in this field.

For LABEL objects, the *equateList* field forms a linked list of unresolved forward references. After compiling the current PROGRAM or UNIT, HLA will traverse this list to resolve any outstanding equates (or emit an unresolved label message).

All other symbols store a NULL in this field.

owner

This pointer contains the address of the object whose sub-symbol table holds this symbol table entry. Usually, this is the address of a procedure, program, unit, method, iterator, or namespace symbol table entry. For the special case of lex level zero symbol table entries, this field contains NULL.

lexLevel

This field holds the static procedure nesting level for the symbol. Note that lex level zero corresponds to global program/unit symbols. Procedures declared in a program or a unit have lex level one, procedures nested in these procedures have lex level two, etc.

objectSize

The *objectSize* field specifies the size, in bytes, of this particular object. If an "object size" doesn't make sense for this object (e.g., for macros or procedures) then this field's value is irrelevant and will probably contain zero. If the object is an array, record, or other composite structure, then this field contains the total size of the object in bytes.

pType

The *pType* field is an enumerated data type that lets the HLA compiler quickly and easily work with predefined data types (e.g., *int32*, *uns8, float64*, and *byte*). The compiler uses this field (along with symClass) to determine how to interpret the *symType* (and related) pointer.

symClass

This field holds the symbol's classification, that is, whether the symbol is a constant, type, variable, procedure, etc. This is one of the following values: *Constant_ct, Value_ct, Type_ct, Var_ct, Parm_ct, Static_ct, Label_ct, Proc_ct, Iterator_ct, ClassProc_ct, ClassIter_ct, Method_ct, Macro_ct, Keyword_ct, Terminator_ct, Program_ct, Namespace_ct, Segment_ct, Register_ct,* and *None_ct*. Along with pType, symClass determines which (if any) of the symbol table pointers HLA should use.

Table 1: Symbol Table Pointer Selection

symClass	Scalar Data Types[a]	Array Types	Record, Union, and Class Types	Pointer Types	Program, Unit, Procedure, Method, Iterator, and Procedure Pointer Types	Namespace and Segment Types	Other Types
Constant_ct, Value_ct, Type_ct, Var_ct, Parm_ct, Static_c	symType	baseType	inhType	baseType	NULL	NULL	NULL

Table 1: Symbol Table Pointer Selection

symClass	Scalar Data Types[a]	Array Types	Record, Union, and Class Types	Pointer Types	Program, Unit, Procedure, Method, Iterator, and Procedure Pointer Types	Namespace and Segment Types	Other Types
Label_ct	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Program_ct, Unit_ct, Proc_ct, Iterator_ct, ClassProc_ct, ClassIter_ct, Method_ct	NULL	NULL	NULL	NULL	localSyms	NULL	NULL
Macro_ct, Keyword_ct, Terminator_ct	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Namespace_ct, Segment_ct	NULL	NULL	NULL	NULL	NULL	localSyms	NULL
Register_ct	NULL	NULL	NULL	NULL	NULL	NULL	NULL

a. boolean, charater, string, enum, uns, int, byte/word/dword,qword/tbyte/lword, and real.

isExternal

This field contains true if the symbol has an external declaration and no local declaration (making the symbol public) for the symbol has been found.

isForward

This field contains true if there has been a forward declaration for the symbol. It is reset to false when the actual declaration appears.

isPrivate

This field contains true if the symbol is a private field of the current data structure (e.g., class or record).

pClass

This field is only valid if the *symClass* field contains *Parm_ct* (meaning that this symbol is a parameter declaration for some program unit). The value of this field determines the parameter's class (that is, the parameter passing

mechanism this particular parameter uses). This field contains one of the *parmClass_t* enumeration values: *valp_pc, refp_pc, vrp_pc, result_pc, name_pc,* or *lazy_pc*.

externName

This field contains the external name for objects. The external name is the name that would be passed to an assembler or linker. For external objects, this is usually the same as *trueName* (unless, of course, the programmer specifies some explicit external name). For non-external objects, this field usually contains an HLA-synthesized name that is guaranteed to be unique. Some objects (e.g., constants and VAR objects) do not have a static name, this field will probably contain NULL for such objects.

offset

For automatic variables (VAR symbols), this field contains the offset into the activation record for the variable. For RECORD and UNION fields, this field contains the offset into the data structure for that field. For VAR objects and METHOD/ITERATOR pointers within CLASSes, the *offset* field contains the offset into the class data structure. Labels, procedures, and (non-class) iterators use this field to hold a unique integer value that HLA's code generators can use to produce a unique label in an assembly or listing output file. Most other objects don't use this field.

Note: the following fields are part of a union and their use is mutually exclusive.

v (type value_t)

If the current symbol is some constant type (*Constant_ct* or *Value_ct*) then this field holds the value associated with that constant symbol. If the constant is an array or record type, then the *v.arrayValues* or *v.fieldValues* fields contain a pointer to the actual data. If the constant is a string or unicode string (*ustring*) value, then the field points at an appropriate HLA string. Otherwise, the value is contained within the *v* field itself.

fields, publicFields, privateSize, fieldCnt, numElements

These fields are used by composite objects (array, records, classes, and unions).

fields

If the current symbol table entry is a type definition and the type is a record, union, or class type, then this field points at a local symbol table that lists the fields of the record. The linear list of this local symbol table specifies the symbols in the order they were declared within the record, union, or class. This field is not valid for non-type symbol table entries.

fieldCnt

If the current symbol table entry is a type definition and the type is a record, union, or class type, then this field specifies the number of fields in that type. This field will contain zero if the current symbol is not a type symbol or is not a record, union, or class declaration.

privateFields

If the current symbol table entry is a type definition and the type is a record, union, or class type, then this field specifies the start of the linear symbol table list that is to be copied whenever some other structure inherits the fields of this symbol. The *fields* pointer points at the start of the linear list for this structure, the *privateFields* pointer points at the first private field of this structure. If *privateFields* contains NULL, then this structure doesn't have any private fields.

numElements

This field contains zero if the object is a scalar object. It contains a positive value if the object is an array object. In the event the object is an array object, the *symType* field contains a pointer to the base type of the array. Note that the *pType* field will contain *Array_pt* if this is an array object. Note that HLA only sets this field to a non-zero value for type declarations. A variable of an array type always has a *numElements* value of zero and the *symType* field points at a data type symbol table entry whose *numElements* field is non-zero.

returnsStr, parms, lastLocal, baseClass, parmSize, callSeq, hasFrame, hasDisplay, alignsStack

These fields are only used by procedures, methods, iterators, pointers to these objects, namespaces, programs, and units.

returnsStr

This field holds a string that is the RETURNS value associated with a procedure or method. Whenever HLA encounters a procedure call in an expression or operand, it substitutes this string for the call in the source file after emitting the code for the call to the procedure. This field is only valid for procedure and method symbols.

parms

This field, which is only valid for procedures, methods, and iterators, points at a linear list of symbols. These symbols are the parameter list for the program unit. If this field is NULL then the program unit doesn't have any parameters. Note that this pointer points at the first (leftmost) parameter in the list and the linear list links the parameters together in the order they are declared. This is true even if the calling sequence is not Pascal.

lastLocal

This field points at the last entry in the linear list of symbols for this object. The symbol table routines use this pointer when adding new symbols to the linear list. his field is used by programs, units, namespaces, procedures, iterators, and methods.

baseClass

For methods, class iterators, and class procedures, this field points at the base class for the program unit. For other objects this field contains NULL. Note that this pointer, if non-null, always points at an object that is a type definition.

parmSize

This field specifies the size of the parameter list (used, for example, by the RET instruction when returning from a procedure to clean up the parameters).

callSeq

For program units, this field specifies the calling sequence (Pascal/HLA, C, StdCall) and is one of the following constant values: *pascal_cs*, *stdcall_cs*, *cdecl_cs*.

hasFrame

If this field contains true, then HLA will emit code at the beginning and end of the program unit to construct and destroy an activation record. This variable also controls code emission for the RETURN instruction. If false, RETURN simply emits a RET instruction; if true, RETURN emits a jump to the code that cleans up the activation record and returns from the procedure.

hasDisplay

This field is true if the program unit is to allocate storage for a display in its activation record. If *hasFrame* and *hasDisplay* are both true, then HLA will actually emit the code to build the activation record.

alignsStack

If this field is true and *hasFrame* is also true, then HLA emits code to double word align the stack after constructing the activation record. This is primarily useful if the caller pushes some number of bytes that is not an even multiple of four prior to calling the current program unit.

Symbol Table Entries for Composite Data Types

Array, structure, and pointer types are always *Type_ct* objects. That is, any time you declare a constant or variable array or structure object, the *symType* field always points at a type definition. Variables and constants are never created directly as array objects. Instead, their symType field points at a symbol table entry that is a type declaration of the appropriate composite type. This is obviously true when you have declarations like the following:

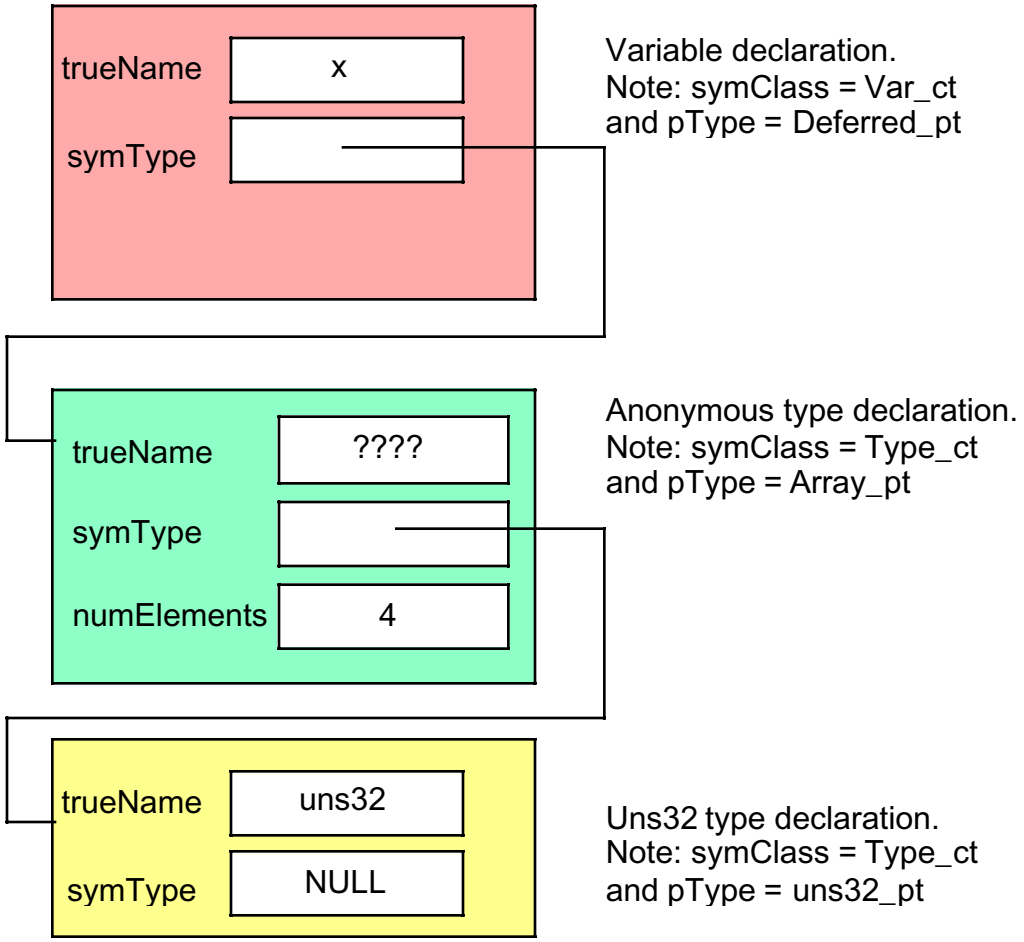
```
type
      a: uns32[4];
      r: record a:uns32; b:uns32; c:uns32; endrecord;

var
      v: a;
      w: r;
```

In this example, *v's symType* field clearly points at the symbol table entry for a while *w's symtype* field obviously points at *r's* entry. What may not be so obvious is the fact that *symType* even points at a type declaration in the following declarations:

```
var
      x: uns32[4];
      y: record a:uns32; b:uns32; c:uns32; endrecord;
```

What is not clear here is exactly *what* symbol table entries the *symType* fields point at. As it turns out, when you have declarations like the ones immediately above, HLA automatically creates an *anonymous* type entry in the symbol table. The normal symbol table search routines will never match these anonymous entries; however the *symType* fields can properly refer to these entries so that the type checking mechanisms in HLA can operate in a consistent manner without a lot of extra kludging. Note that the *pType* fields for *x* and *y* will contain the value *Deferred_pt*. This tells HLA that *x* and *y* are composite types and it should follow the *symType* pointer to determine their type.



Note that HLA's symbol table declaration only provides a single *numElements* field for specifying the number of elements in an array object. This means that the symbol table format supports only a single dimension array. A good question is "How does HLA's symbol table structure support multi-dimensional array declarations?" For example, the following is a perfectly legal HLA declaration:

```
var
      a2: uns32[8,4];
```

To understand how HLA implements the above two-dimensional array declaration, consider the following declaration:

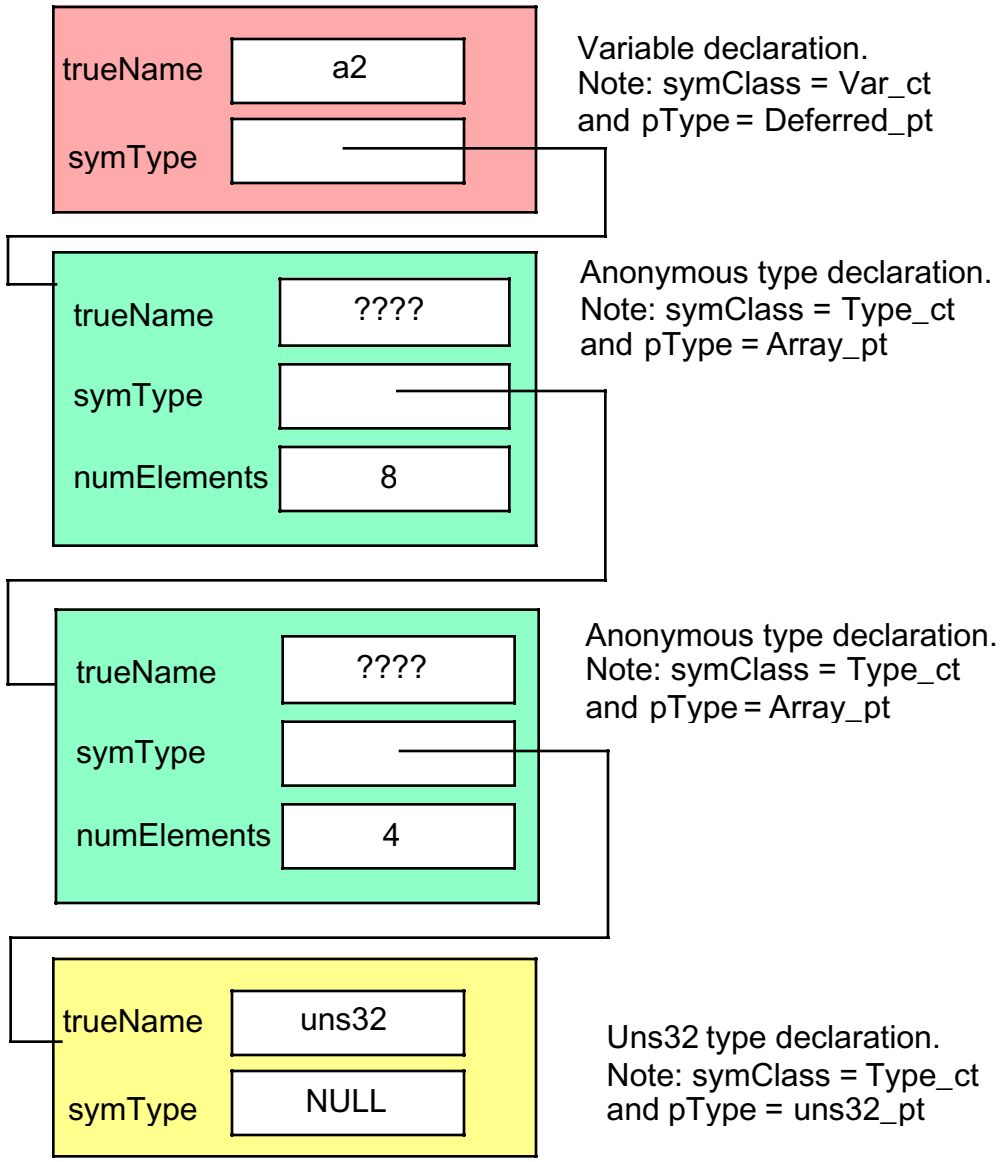
```
type
      u1: uns32[4];
```

```
        u2: u1[8];

var
        a3: u2;
```

In this example, it's easier to see how HLA implements multi-dimensional arrays using an "array of arrays" mechanism. Clearly, it's easy to implement the *u1* type; all HLA has to do is set the *numElements* field to four and point the *symType* field at the *uns32* symbol table entry. Logically, implementing *u2* is no different. HLA sets *u2 s numElements* field to eight and points the *symType* field at the symbol table entry for *u1*.

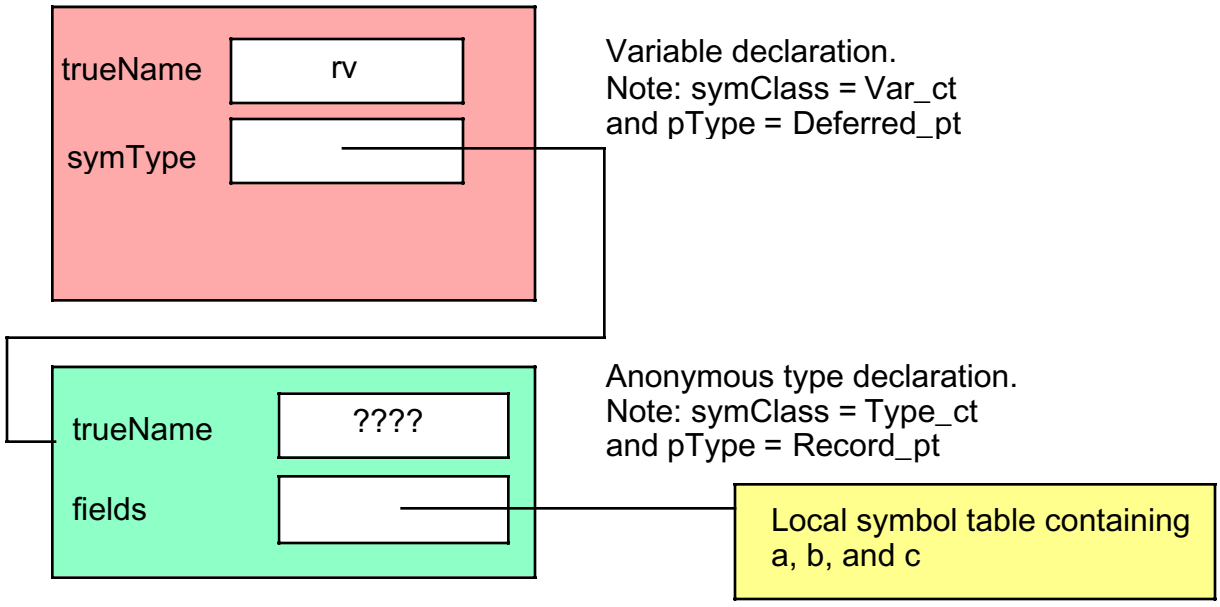
When HLA encounters a multi-dimensional array declaration like the one for *a2* above, it creates an anonymous type declaration for each of the dimensions in the declaration. Therefore, *a2 s symType* field will wind up pointing at an anonymous type entry whose *numElements* field contains eight and whose *symType* field points at a second symbol table type entry. That second anonymous entry will contain four in its *numElements* field and the *symType* field will point at the *uns32* symbol table entry.



HLA handles direct (anonymous) record declarations in an identical fashion. If you have a record variable declared as follows:

```
var
      rv:record
            a:uns32;
            b:uns32;
            c:uns32;
      endrecord;
```

then HLA will create a symbol table entry for *rv* that points at a symbol table entry that HLA creates on the fly. The new (anonymous) symbol table entry will have a fields field that points at the entries for *a*, *b*, and *c*, have a *fieldCnt* value of three, etc. Note that the *pType* field for *rv* will contain the value *Deferred_pt*. This tells HLA that *rv* is a composite type and it should follow the *symType* pointer to determine *rv s* type.



Note that this use of anonymous symbol table entries is a big departure from the technique used in HLA v1.x. HLA v1.x did not use anonymous symbol table entries (which is a standard compiler symbol table management trick) and, instead, implemented structures and array directly in the variable's symbol table entry. While there were some minor benefits to the way HLA v1.x handled these entries, the disadvantages far outweighted the benefits. The new (well, traditional actually) scheme employed by HLA v2.0 is much simpler and probably more efficient in the long run[1].

Pointers are another composite data type that HLA handles in a fashion just like arrays and structures. Consider the following declaration:

```
var
      p: pointer to dword;
```

1. One might ask if the traditional scheme is so good and so well known, why didn't HLA v1.x use this scheme. The reason is that HLA v1.x was a prototype and several experiments were tried in the code. The mechanism for implementing arrays and records was one such experiment that, alas, failed.

The symbol table entry for *p* will contain *Deferred_pt* as the *pType* value and *symType* will point at an anonymous type entry. That anonymous entry will have a *pType* value of *Pointer_pt* and the *baseType* field will point at the symbol table entry for *dword*.

Symbol Table Data for Specific Symbol Classes

The following subsections describe the meaning of the symbol table entries for each of the symbol classes:

LABEL Class Symbol Table Entries

Label objects let HLA programmers forward declare statement labels in an HLA program. Statement labels are NEAR32 symbols (using MASM terminology). The principle purpose for a LABEL object is to allow code in one procedure to reference a label declared in a separate procedure. Note that a procedure may forward reference any local statement labels appearing in that procedure; the purpose of the LABEL section is to declare symbols that other procedures may reference.

Here is the syntax for the label section (note: optional items are underlined):

```
label
      identifier; option
      << Additional label identifier declarations >>
```

Allowable options (these are mutually exclusive):

```
            external;
            external( "extern_name" );
```

Example:

```
      label
            aLabelID;
            AnExternalLbl; external;
            AnotherExtLbl; external( "external_name" );
```

Note: All labels you declare in a procedure must appear as statement labels within the current procedure or some other procedure must define a global symbol using that label. To define a statement label in a program, you simply put the label's name followed by a colon anywhere a statement is legal, e.g.,

```
      LabelExample:
            mov( 0, eax );
```

By default, labels are local to the procedure in which you define them. You may define global labels by following the identifier with a pair of colons, e.g.,

```
      GlobalLabel::
            mov( 1, ebx );
```

Global symbols must be unique from the current lex level up to lex level zero (that is, there cannot be a visible symbol identical to the global label at the point you define the global label) unless that label was defined somewhere in a LABEL declaration section (in which case the global label declaration satisfies the requirement that the symbol must be defined at some point. By declaring a label at lex level one (the program/unit lex level), you can easily refer to a

label inside any procedure from within any other procedure (though it's probably not good programming style to do so).

```
program NestedLabelReference;
label
      nestedLabel;

      procedure UsesNestedLabel;
      begin UsesNestedLabel;

            jmp nestedLabel; // Not bright, just for demo only.

      end UsesNestedLabel;

      procedure DefinesNestedLabel;
      begin DefineNestedLabel;

            nestedLabel::  // Satisfies nestedLabel declaration earlier.

      end DefineNestedLabel;

begin NestedLabelReference;

      ...

end NestedLabelReference;
```

Like other HLA public declarations, you create public labels by both declaring the label to be external and then defining the label in the current source file. For example, we can make "nestedLabel" accessible from other source files in the example above by changing the label declaration to the following:

```
label
      nestedLabel; external;
```

Here's how LABEL symbol table entries use the fields in the symbol table:

lcname, truename

These symbol table fields contain the declared name of the symbol.

equateLabel

If the current label is equated to another label, and that label has not yet been defined, then this field points at the symbol table entry for that other label. Once the label is defined, HLA sets this field to NULL. If the current label object is not equated to another label, then this field will be NULL.

seg

This field contains NULL for label objects.

equateList (segList)

This field forms a linked list of labels that are equated to other labels. After processing a PROGRAM or a UNIT, HLA scans through this equate list to determine if there are any unresolved label declarations and to resolve any outstanding equated labels.

owner

This field points at the object whose localSymbols field contains a pointer to the sub-symbol-table that holds this symbol (typically a PROGRAM, UNIT, PROCEDURE, ITERATOR, or METHOD symbol table entry).

lexLevel

This field holds the numeric lex level of this symbol.

objectSize

This field has no meaning for label objects.

pType

The field contains the constant *Label_pt* for label objects.

symClass

This field contains *Label_ct* for label objects.

isExternal

This field contains true if the symbol is externally defined (i.e., the EXTERNAL clause is present).

isForward

This field is always set to true when you define a symbol in the LABEL section. HLA sets this field to false when it encounters the actual symbol definition in the procedure.

isPrivate

This field has no meaning for label objects.

pClass

This field has no meaning for label objects.

externName

This field contains the external name of the label object. If this symbol is not external, then this field contains NULL.

address (offset)

HLA stores the address of the intermediate code associated with the label in this field. At code generation time, the compiler uses this information to compute the offset of some label into the program's code space.

Label declarations do not use any other fields in the symbol table data structure. Also note that labels may only appear as global symbols in programs/units or local symbols in procedures, iterators, and methods. They may not appear in namespace or structure declarations.

enterLabel Function Prototype:

```
procedure enterLabel
(
      symbol: string;
      staticName: string;
      address: ptrIntCode;
      owner: symNodePtr_t
);    returns( "eax" );
```

This procedure inserts a label symbol into the current symbol table. If it succeeds, it returns a pointer to the new symbol table entry in the EAX register. If it fails, it returns NULL (zero) in EAX; if it fails, the function will print the appropriate error message, the caller need not take any special action. Failure usually implies a duplicate symbol error, though there are a few other degenerate cases the function handles, as well.

The *symbol* parameter must point at an HLA string that contains the label's identier. This field must not be NULL and must point at a non-empty string. HLA assumes that this particular string is not shared with any other data object. Therefore, the caller must first duplicate this string if some other long-term data structure will also point at this string data.

The *staticName* parameter will point at an HLA string if this is an EXTERNAL (or public) label declaration. If the EXTERNAL keyword appears by itself, this string should be a duplicate of the symbol parameter (but it should not be the same exact string). If the optional string parameter follows the EXTERNAL keyword, then the *staticName* parameter should be the string specified as the EXTERNAL string.

The *address* parameter contains the memory address of the intermediate code associated with the statement to which the label applies. When processing symbols in the LABEL section, this offset generally isn't known yet, so HLA simply passes zero in this parameter. HLA's code generator uses this address to determine jump offsets and other such information at code generation time.

The *owner* parameter contains a pointer to the object whose sub-symbol table contains this label.

CONST Class Symbol Table Entries

CONST class symbols have a (compile and run time) immutable value associated with them. HLA binds the value of a CONST object to the symbol at the point of declaration and that value never changes thereafter.

lcname, truename

These symbol table fields contain the declared name of the symbol.

symType

This field contains a pointer to the symbol's data type.

seg, segList

These fields contain NULL for CONST objects.

owner

This field points at the object whose *localSymbols* field contains a pointer to the sub-symbol-table that holds this symbol.

lexLevel

This field holds the numeric lex level of this symbol.

objectSize

This field holds the size of the constant, in bytes.

pType

The field contains the enumerated *pType_t* constant associated with this constant's data type. Usually the value is something like *Uns32_pt*, although HLA constants can be composite as well as scalar data types. For composite and user-defined types, the *pType* field will contain *Deferred_pt*. In that case, the *symType* field will point at an appropriate symbol table entry (usually an array, record, or pointer type entry, since these are the only composite types HLA allows as constants).

symClass

This field contains *Constant_ct* for constant objects.

isExternal

This field always contains false for CONST objects.

isForward

This field always contains false for CONST objects.

isPrivate

This field contains true if this CONST symbol is a private field of a CLASS object.

pClass

This field has no meaning for CONST objects.

externName

This field has no meaning for CONST objects and contains NULL.

Variant Data Fields (*v*)

One of the variant data fields (e.g., *v.uns32_vt*) holds the value associated with this constant symbol. The *pType* and *symType* fields determine which of the variant data fields holds the value. For array and record types, the *arrayValues_vt* or *fieldValues_vt* fields contain the address of the constant data elsewhere in memory. HLA does not support union or class constants.

enterConst Function Prototype:

```
procedure enterConst
(
            symbol      :string;
      var   constVal    :attr_t;
            owner       :symNodePtr_t
);    returns( "eax" );
```

This procedure inserts a CONST symbol into the symbol table specified by *owner* (into the current symbol table if *owner* is NULL). If it succeeds, it returns a pointer to the new symbol table entry in the EAX register. If it fails, it returns NULL (zero) in EAX; if it fails, the function will print the appropriate error message, the caller need not take any special action. Failure usually implies a duplicate symbol error, though there are a few other degenerate cases the function handles, as well.

The *symbol* parameter must point at an HLA string that contains the constant's identifier. This field must not be NULL and must point at a non-empty string. HLA assumes that this particular string is not shared with any other data object. Therefore, the caller must first duplicate this string if some other long-term data structure will also point at this string data.

The *constVal* parameter must be a pointer to the value to assign to this constant symbol. HLA makes a copy of the *value_t* component of *constVal*, so the caller doesn't have to preallocate the data for the *value_t* data. However, if there are any pointers to auxiliary data (e.g., strings, arrays, or records), then HLA assumes that such data has been allocated on the heap and no other data structures refer to this data (that is, the pointer contained within the *attr_t* structure is the only pointer that refers to this data). If this is not the case, then *enterConst s* caller must duplicate this data prior to calling *enterConst*. Note that most of the time this structure is filled in by the expression evaluator and the expression evaluator guarantees unique copies of the data.

The owner parameter points at the object (procedure, namespace, etc.) whose sub-symbol table will hold this constant's symbol. If this field contains NULL then HLA inserts the symbol into the currently active symbol table.

VAL/? Class Symbol Table Entries

VAL class symbols have run time immutable values associated with them, though their values can change at compile-time (e.g., by executing compile-time language statements like "?"). HLA binds the value of a VAL object to the symbol at the point of declaration and that value remains constant until explicitly changed by the "?" compile-time statement.

There is a big semantic difference between declaring (and assigning a value to) a symbol in the VAL section versus using the "?" compile-time statement. A statement like the following:

```
? XYZ : uns32 := 0;
```

will use a previously-declared *XYZ* symbol, even if that symbol was defined at a different lex level. This statement can change both the type of value of the previously defined symbol. If *XYZ* was not previously defined, then the "?" statement above declares a new symbol at the current scope. This statement never generates an error, regardless of whether *XYZ* was previously declared at the same or a different lex level (or not at all).

Contrast the behavior above with the following *Val* declaration:

```
VAL
      XYZ : uns32 := 0;
```

This declaration will replace the value of symbol *XYZ* if it already exists at the current lex level. If the symbol *XYZ* exists at a different lex level, then the declaration above does not affect the value or type of the original symbol; instead, this declaration creates a new copy of the symbol at the current lex level.

VAL and "?" symbols use the fields of a symbol table entry in exactly the same manner as CONST objects. The only difference between the two is that the *symClass* field contains *Value_ct* and it's possible to change the symbol's type by replacing the value of the *v* field (and the associated type information).

enterVal Function Prototype:

The *enterVal* function enters a symbol into a symbol table whose declaration appears in a VAL section. This function looks only in the local symbol table specified by owner (or the currently active symbol table if *owner* is NULL). If it doesn't find the symbol in the local symbol table, it creates that symbol and assigns the associated value; if it does find the symbol, and that symbol's class is *Value_ct*, then HLA frees the current value associated with that symbol and assigns the value of *constVal* to the symbol. The parameters for *enterVal* are identical to those for *enterConst*.

```
procedure enterVal
(
            symbol      :string;
      var   constVal    :attr_t;
            owner       :symNodePtr_t
);    returns( "eax" );
```

setVal Function Prototype:

The *setVal* function handles the "?" operator. This function first checks to see if the symbol is currently visible at any lex level; if so, and that symbol's class is *Value_ct*, then HLA frees the current value associated with that symbol and assigns the value of *constVal* to the symbol. If the specified symbol is not visible at the point the "?"

operator appears, then *setVal* behaves identically to *enterVal*. The parameters for *enterVal* are identical to those for *enterConst*.

```
procedure setVal
(
            symbol      :string;
      var   constVal    :attr_t;
            owner       :symNodePtr_t
);    returns( "eax" );
```

TYPE Symbol Table Entries

The TYPE section lets a program define new data types in HLA. The syntax for a TYPE declaration takes one of the following forms (optional items are underlined):

```
type
      id : typeID;                  // Isomorphism (type renaming).
      id : enum{ id_list };         // Create an enumerated data type.
      id : pointer to typeID;       // Creates a pointer type to a base type.
      id : typeID[ dimlist ];       // Creates an array type.
      id : record fields endrecord; // Creates a record type.
      id : union fields endunion;   // Creates a union type.
      id : class decls endclass;    // Creates a class type.
      id : procedure( parms );      // Creates a procedure pointer type.
      id : iterator( parms );       // Creates an iterator pointer type.
```

The following is legal in the TYPE declaration section, but it doesn't create an actual TYPE symbol table entry.

```
      id : forward(id);       // Forward declaration for use by macros.
```

Here's how TYPE symbols use the symbol table fields:

lcname, truename

These symbol table fields contain the declared name of the symbol.

symType, baseType, inhType (depends on declaration)

For primitive types (e.g., *int32*) this field contains NULL (zero). For isomorphic types (type renaming) this field contains a pointer to the type entry whose type this definition is renaming. For array types, the *baseType* field points at the type field for an array element. For structure types that inherit fields from another structure, the *inhtype* field contains a pointer to the base type. For most other types, this field contains NULL.

seg, segList

These fields contain NULL for TYPE objects.

owner

This field points at the object whose *localSymbols* field contains a pointer to the sub-symbol-table that holds this symbol.

lexLevel

This field holds the numeric lex level of this symbol.

objectSize

This field holds the size the data type will consume, in bytes. For composite types, this is the total number of bytes associated with the object. Note that string and ustring objects are pointers, hence this field contains four. This field has no meaning for procedures, iterators, methods, programs, macros, and TEXT objects. Note that this field does contain four for pointers to these objects.

pType

The field contains the enumerated *pType_t* constant associated with this constant's data type. Usually the value is something like *Uns32_pt*, although HLA constants can be composite as well as scalar data types. For composite and user-defined types, the *pType* field usually contains something like *Array_pt* or *Record_pt*.

Boolean_pt:	Associated with type boolean and isomorphisms of boolean. *SymType* field contains NULL.
Enum_pt:	Associated with enumerated types. SymType field contains NULL.
Uns8_pt:	Associated with type uns8 and isomorphisms of uns8. *SymType* field contains NULL.
Uns16_pt:	Associated with type uns16 and isomorphisms of uns16. *SymType* field contains NULL.
Uns32_pt:	Associated with type uns32 and isomorphisms of uns32. *SymType* field contains NULL.
Uns64_pt:	Associated with type uns64 and isomorphisms of uns64. *SymType* field contains NULL.
Uns128_pt:	Associated with type uns128 and isomorphisms of uns128. *SymType* field contains NULL.
Byte_pt:	Associated with type byte and isomorphisms of byte. *SymType* field contains NULL.
Word_pt:	Associated with type word and isomorphisms of word. *SymType* field contains NULL.
DWord_pt:	Associated with type dword and isomorphisms of dword. *SymType* field contains NULL.
QWord_pt:	Associated with type qword and isomorphisms of qword. *SymType* field contains NULL.
TByte_pt:	Associated with type tbyte and isomorphisms of tbyte. *SymType* field contains NULL.
LWord_pt:	Associated with type lword and isomorphisms of lword. *SymType* field contains NULL.
Int8_pt:	Associated with type int8 and isomorphisms of int8. *SymType* field contains NULL.
Int16_pt:	Associated with type int16 and isomorphisms of int16. *SymType* field contains NULL.
Int32_pt:	Associated with type int32 and isomorphisms of int32. *SymType* field contains NULL.
Int64_pt:	Associated with type int64 and isomorphisms of int64. *SymType* field contains NULL.
Int128_pt:	Associated with type int128 and isomorphisms of int128. *SymType* field contains NULL.
Char_pt:	Associated with type char and isomorphisms of char. *SymType* field contains NULL.
XChar_pt:	Associated with type xchar and isomorphisms of xchar. *SymType* field contains NULL.
Unicode_pt:	Associated with type unicode and isomorphisms of unicode. *SymType* field contains NULL.
Real32_pt:	Associated with type real32 and isomorphisms of real32. *SymType* field contains NULL.

Real64_pt:	Associated with type real64 and isomorphisms of real64. *SymType* field contains NULL.
Real80_pt:	Associated with type real80 and isomorphisms of real80. *SymType* field contains NULL.
String_pt:	Associated with type string and isomorphisms of string. *SymType* field contains NULL.
UString_pt:	Associated with type ustring and isomorphisms of ustring. *SymType* field contains NULL.
Cset_pt:	Associated with type cset and isomorphisms of cset. *SymType* field contains NULL.
XCset_pt:	Associated with type xcset and isomorphisms of xcset. *SymType* field contains NULL.
Thunk_pt:	Specifies a thunk type. *SymType* is NULL.
Deferred_pt:	Used for type isomorphisms (type renaming). Tells HLA to get type from *symType* field.
Array_pt:	Specifies an array type. *SymType* points at the base type.
Record_pt:	Specifies a record type. Base points at fields list. *SymType* field contains NULL.
Union_pt:	Specifies a union type. Base points at fields list. *SymType* field contains NULL.
Class_pt:	Specifies a class type. Base points at fields list. *SymType* field contains NULL.
Pointer_pt:	Specifies a pointer type. *baseType* points at the base type.
Procptr_pt:	Specifies a procedure pointer type. *SymType* field contains NULL. The procedure related fields contains other procedure data.
AnonRec_pt:	Specifies an anonymous record type. Generally not used by definitions in the TYPE section.
Namespace_pt:	Specifies a namespace type. Generally not used by definitions in the TYPE section.
Segment_pt:	Specifies a segment type. Generally not used by definitions in the TYPE section.
Label_pt:	Specifies a statement label type. *SymType* is NULL.
Proc_pt:	Specifies a procedure type. *SymType* field contains NULL. The procedure related fields contains other procedure data.
Method_pt:	Specifies a method type. *SymType* field contains NULL. The procedure related fields contains other procedure data.
ClassProc_pt:	Specifies a class procedure type. *SymType* field contains NULL. The procedure related fields contains other procedure data.
ClassIter_pt:	Specifies a class iterator type. *SymType* field contains NULL. The procedure related fields contains other procedure data.
Iterator_pt:	Specifies an iterator type. *SymType* field contains NULL. The procedure related fields contains other procedure data.
Program_pt:	Specifies a program type. Generally not used by definitions in the TYPE section (only legal for the program identifier).
Macro_pt:	Specifies a macro type. Generally not used by definitions in the TYPE section.
Text_pt:	Specifies a TEXT type. SymType field contains NULL.
Variant_pt:	Specifies a variant data type. Generally not used by definitions in the TYPE section.
Error_pt:	This type is never used in the symbol table. It is returned as a *pType* value to indicate some sort of error during parsing.

symClass

This field contains *Type_ct* for TYPE objects.

isExternal, isForward

These fields don't apply to TYPE objects, so they always contain false.

isPrivate

This field contains true if this type symbol is a private field of a CLASS object.

pClass

Not used by TYPE objects.

externName

This field generally contains NULL.

offset/address

Not used by TYPE objects.

variant fields

Not used by TYPE objects.

numElements

For array objects, this field contains the number of elements for the array. For scalar objects, this field contains zero.

fields

For record, union, and class objects, this field points at the linear list of fields associated with the structure. For other types this field contains NULL.

privateFields

For record, union, and class objects, this field points at the linear list of private fields associated with the structure. For other types this field contains NULL.

fieldCnt

For record, union, and class objects, this field contains the number of fields in the structure. For other types, this field contains zero.

Procedure Related Fields (returnsStr, parms, etc.)

These fields have meaning if the *pType* field holds *Procptr_pt, Proc_pt, Method_pt, ClassProc_pt, ClassIter_pt,* or *Iterator_pt*. In this case, these fields contain the same information as you would find for a procedure symbol table entry (see "PROGRAM, UNIT, PROCEDURE, METHOD, and ITERATOR Symbol Table Entries" on page 38 for more details).

enterType Function Prototype

The enterType procedure inserts a TYPE symbol into the symbol table. Here's the prototype for this procedure:

```
procedure enterType
(
      symbol      :string;
      pType       :pType_t;
      theType     :symNodePtr_t;
      owner       :symNodePtr_t
);    returns( "eax" );
```

The parameters have their usual meanings. Note that *theType* corresponds to the *symType/baseType/inhType* field (depending upon the value of *pType*). Note that this procedure does not fill in the values for RECORD, ARRAY, UNION, CLASS, and PROCEDURE/METHOD/ITERATOR types. The caller must do this after *enterType* returns (using the pointer to the symbol table entry that *enterType* returns in EAX).

VAR Symbol Table Entries

The VAR section lets a program define automatic variables in HLA. The syntax for a VAR declaration takes one of the following forms:

```
type
      id : typeID;                  // Simple declaration.
      id : enum{ id_list };         // Create an enumerated variable.
      id : pointer to typeID;       // Creates a pointer variable.
      id : typeID[ dimlist ];       // Creates an array variable.
      id : record fields endrecord; // Creates a record variable.
      id : union fields endunion;   // Creates a union variable.
      id : procedure {( parms )};   // Creates a procedure pointer var.
```

The following is legal in the VAR declaration section, but it doesn't create an actual VAR symbol table entry.

```
      id : forward(id);       // Forward declaration for use by macros.
```

Here's how VAR symbols use the symbol table fields:

lcname, truename

These symbol table fields contain the declared name of the symbol.

symType

This field contains a pointer to the symbol table entry that corresponds to this variable's type.

seg, segList

These fields contain NULL for VAR objects.

owner

This field points at the object whose *localSymbols* field contains a pointer to the sub-symbol-table that holds this symbol.

lexLevel

This field holds the numeric lex level of this symbol.

objectSize

This field holds the size the variable will consume, in bytes. For composite types, this is the total number of bytes associated with the variable. Note that string and ustring objects are pointers, hence this field contains four. This field has no meaning for procedures, iterators, methods, programs, and macros. Note that this field does contain four for pointers to these objects.

pType

The field contains the enumerated *pType_t* constant associated with this variable's data type. Usually the value is something like *Uns32_pt*, although HLA variables can be composite as well as scalar data types. For composite and user-defined types, the *pType* field usually contains *Deferred_pt*.

Boolean_pt:	Object is a boolean variable.
Enum_pt:	Object is a variable that is an enumerated type.
Uns8_pt:	Object is an uns8 variable.
Uns16_pt:	Object is an uns16 variable.
Uns32_pt:	Object is an uns32variable.
Uns64_pt:	Object is an uns64 variable.
Uns128_pt:	Object is an uns128 variable.
Byte_pt:	Object is a byte variable.
Word_pt:	Object is a word variable.
DWord_pt:	Object is a dword variable.
QWord_pt:	Object is a qword variable.
TByte_pt:	Object is a tbyte variable.
LWord_pt:	Object is an lword variable.
Int8_pt:	Object is an int8 variable.
Int16_pt:	Object is an int16 variable.
Int32_pt:	Object is an int32 variable.

Int64_pt:	Object is an int64 variable.
Int128_pt:	Object is an int128 variable.
Char_pt:	Object is a char variable.
XChar_pt:	Object is an xchar variable.
Unicode_pt:	Object is a unicode character variable.
Real32_pt:	Object is a real32 variable.
Real64_pt:	Object is a real64 variable.
Real80_pt:	Object is a real80 variable.
String_pt:	Object is a string variable.
UString_pt:	Object is a unicode string variable.
Cset_pt:	Object is a character set variable.
XCset_pt:	Object is an extended character set variable.
Thunk_pt:	Object is a thunk variable.
Deferred_pt:	Used for composite types. Tells HLA to get type from *symType* field.

`The other "_pt" values don't apply to VAR objects.`

symClass

This field contains *Var_ct* for VAR objects or *Parm_ct* for p arameters (which are also considered VAR objects).

isExternal, isForward

These fields always contain false for VAR objects.

isPrivate

This field contains true if this VAR object appears in the private section of a class definition.

pClass

If this symbol table entry is a parameter object, then this field contains the parameter passing mechanism. This is one of the following values: *valp_pc, refp_pc, vrp_pc, result_pc, name_pc*, or *lazy_pc*.

externName

This field contains NULL.

offset

Contains the offset into the activation record for this VAR object.

The remaining fields have no meaning in a VAR declaration symbol table entry. For composite data types and procedure variables, HLA always creates an anonymous type entry rather than use these fields within the variable's symbol table entry.

enterVal Function Prototype

The following function inserts a var object into the symbol table:

```
procedure enterVar
(
      symbol      :string;
      pType       :pType_t;
      symType     :symNodePtr_t;
      owner       :symNodePtr_t;
      offset      :int32;
      pClass      :parmClass_t
);    returns( "eax" );
```

The symbol, pType, symtype, and owner parameters have the usual meaning. The offset parameter holds the offset of the variable into the current activation record; this value is positive for parameters and negative for local variables. The pClass parameter is meaningful only for parameters; it holds the parameter passing type of the symbol; the caller should pass *notp_pc* (zero) for non-parameter objects.

STATIC, READONLY, and STORAGE Symbol Table Entries

The STATIC, READONLY, AND STORAGE sections let a program define static variables in HLA. The difference between these sections is whether or not you can initialize the objects at compile time (STATIC and READONLY) and whether you may write to the segment at run-time (STATIC and STORAGE)

The syntax for a STORAGE declaration takes one of the following forms:

```
storage
      id : typeID;                  // Simple declaration.
      id : enum{ id_list };         // Create an enumerated variable.
      id : pointer to typeID;       // Creates a pointer variable.
      id : typeID[ dimlist ];       // Creates an array variable.
      id : record fields endrecord; // Creates a record variable.
      id : union fields endunion;   // Creates a union variable.
      id : procedure {( parms )};   // Creates a procedure pointer var.
```

The following is legal in the STORAGE declaration section, but it doesn't create an actual STORAGE symbol table entry.

```
      id : forward(id);       // Forward declaration for use by macros.
```

The syntax for a READONLY declaration takes one of the following forms:

```
readonly
      id : typeID := expression;
      id : enum{ id_list } := expression;
      id : pointer to typeID := expression;
      id : typeID[ dimlist ] := expression;
      id : record fields endrecord; := expression;
      id : procedure {( parms )} := expression;
```

Note that the types of the above expression must match the type of the object. In particular, pointer and procedure variables require pointer constants, array declarations require array constants, and record declaration require record constants.

The following is legal in the READONLY declaration section, but it doesn't create an actual READONLY symbol table entry.

```
      id : forward(id);       // Forward declaration for use by macros.
```

Semantic Note: if HLA detects an attempt to write to a READONLY variable (e.g., "mov(eax, ROvar);") then HLA will issue a warning during the compilation of the program. This is not a fatal error, but suggests that a run-time error may result when you execute the code. Note that it is not possible for HLA to detect all attempts to write to a read-only object (e.g., HLA will not detect an attempt to take the address of a READONLY variable and then store a value indirectly at that address). HLA does not issue a warning if you attempt to pass a READONLY object by reference to a procedure (since the procedure doesn't have to write to the address you pass).

The STATIC declaration section allows any of the declarations above (initialized or uninitialized). Of course, the static section begins with the reserved word STATIC rather than READONLY or STORAGE:

```
static
      id : typeID;
      id : enum{ id_list };
      id : pointer to typeID;
      id : typeID[ dimlist ];
      id : record fields endrecord;
      id : union fields endunion;
      id : procedure {( parms )};

      id : typeID := expression;
      id : enum{ id_list } := expression;
      id : pointer to typeID := expression;
      id : typeID[ dimlist ] := expression;
      id : record fields endrecord; := expression;
      id : procedure {( parms )} := expression;
```

The following is legal in the STATIC declaration section, but it doesn't create an actual STATIC symbol table entry.

```
      id : forward(id);       // Forward declaration for use by macros.
```

Here's how variables you declare in the STATIC, READONLY, and STORAGE sections use the symbol table fields:

lcname, truename

These symbol table fields contain the declared name of the symbol.

symType

This field contains a pointer to the symbol table entry that corresponds to this variable's type.

seg

The owner field points at the segment that owns this object. For STATIC variables this field always points at the symbol table entry for the "static" segment (the data segment). For READONLY variables, this field always points at the "readonly" segment symbol table entry. For STORAGE variables, this field always points at the "bss" segment symbol table entry.

segList

This pointer forms a linearly linked list of symbols that all belong to the same segment. This will chain all STATIC objects together, all READONLY objects together, and all STORAGE objects together. HLA uses these lists to emit the static sections to the object code during the code generation phase.

owner

This field points at the object whose *localSymbols* field contains a pointer to the sub-symbol-table that holds this symbol.

lexLevel

This field holds the numeric lex level of this symbol.

objectSize

This field holds the size the variable will consume, in bytes. For composite types, this is the total number of bytes associated with the variable. Note that string and ustring objects are pointers, hence this field contains four. This field has no meaning for procedures, iterators, methods, programs, and macros. Note that this field does contain four for pointers to these objects.

pType

The field contains the enumerated *pType_t* constant associated with this variable's data type. Usually the value is something like *Uns32_pt*, although HLA variables can be composite as well as scalar data types. For composite and user-defined types, the *pType* field usually contains *Deferred_pt* (in which case the *symType* field points at a symbol table entry that specifies the type of the variable).

Boolean_pt:	Object is a boolean variable.
Enum_pt:	Object is a variable that is an enumerated type.
Uns8_pt:	Object is an uns8 variable.
Uns16_pt:	Object is an uns16 variable.
Uns32_pt:	Object is an uns32variable.

Uns64_pt: Object is an uns64 variable.
Uns128_pt: Object is an uns128 variable.

Byte_pt: Object is a byte variable.
Word_pt: Object is a word variable.
DWord_pt: Object is a dword variable.
QWord_pt: Object is a qword variable.
TByte_pt: Object is a tbyte variable.
LWord_pt: Object is an lword variable.

Int8_pt: Object is an int8 variable.
Int16_pt: Object is an int16 variable.
Int32_pt: Object is an int32 variable.
Int64_pt: Object is an int64 variable.
Int128_pt: Object is an int128 variable.

Char_pt: Object is a char variable.
XChar_pt: Object is an xchar variable.
Unicode_pt: Object is a unicode character variable.

Real32_pt: Object is a real32 variable.
Real64_pt: Object is a real64 variable.
Real80_pt: Object is a real80 variable.

String_pt: Object is a string variable.
UString_pt: Object is a unicode string variable.
Cset_pt: Object is a character set variable.
XCset_pt: Object is an extended character set variable.

Thunk_pt: Object is a thunk variable.

Deferred_pt: Used for composite types. Tells HLA to get type from *symType* field.

The other "_pt" values don't apply to STATIC, READONLY, and STORAGE objects.

symClass

This field contains *Static_ct* for STATIC, READONLY, and STORAGE objects.

isExternal

This field contains true if this is an external symbol declaration. Note that external declarations are legal only at lex level one.

isForward

The STATIC, READONLY, and STORAGE objects do not use this field. It always contains false for these objects.

isPrivate

This field contains true if the current variable declaration appears in a PRIVATE section of a CLASS declaration.

pClass

If this symbol table entry is a parameter object, then this field contains the parameter passing mechanism. This is one of the following values: *valp_pc, refp_pc, vrp_pc, result_pc, name_pc*, or *lazy_pc*.

externName

This field contains NULL.

address

This field points at the intermediate code generated for the variable's declaration. The code generator uses this information to determine the variable's run-time offset.

v (variant data) Field

For STATIC and READONLY objects, HLA stores the initial value of the object in this field. For uninitialized STATIC objects, this field contains all zeros (and, therefore, the array/record pointer fields will contain NULL). STORAGE objects do not use this field (since you cannot initialize them).

STATIC, READONLY, and STORAGE objects do not use any of the remaining fields in the symbol table record. Procedure variables and composite objects use an anonymous type record, if necessary, to hold the other information.

enterStatic, enterReadonly, enterStorage Function Prototypes

These three functions insert a symbol into the symbol table associated with the specified static segment. The prototypes are the following:

```
procedure enterStatic
(
            symbol      :string;
            externName  :string;
            pType       :pType_t;
            symType     :symNodePtr_t;
            owner       :symNodePtr_t;
      var   v           :value_t
);    returns( "eax" );
```

The *symbol, pType, symType*, and *owner* parameters have the usual values. The *externName* parameter contains NULL if this is not an EXTERNAL symbol, it points at the external name if it is an EXTERNAL symbol. The *symbol* and *externName* parameters cannot both point at the same physical string in memory. The *v* parameter holds the initial value for this STATIC object; this parameter contains NULL if the STATIC object doesn't have an initial value.

```
procedure enterReadonly
```

```
(
            symbol      :string;
            externName  :string;
            pType       :pType_t;
            symType     :symNodePtr_t;
            owner       :symNodePtr_t;
      var   v           :value_t
);    returns( "eax" );
```

The *symbol, externName, pType, symType*, and *owner* parameters have the usual values (see above). The *v* parameter holds the initial value for this READONLY object; this parameter must not contain NULL; it must contain a value constant that is compatible with the symbol's data type.

```
procedure enterStorage
(
            symbol      :string;
            externName  :string;
            pType       :pType_t;
            symType     :symNodePtr_t;
            owner       :symNodePtr_t
);    returns( "eax" );
```

The *symbol, externName, pType, symType*, and *owner* parameters have the usual values.

SEGMENT Symbol Table Entries

The STATIC, READONLY, and STORAGE sections in HLA are special cases of *segments*[1]. Variables you declare in the STATIC section belong to the *data* segment, variables you declare in the STORAGE section belong to the *bss* segment, and variables you declare in the READONLY section belong to the *readonly* segment. HLA allows the creation of additional segments using the SEGMENT keyword. The SEGMENT declaration section uses the following syntax (optional items are underlined):

```
segment segname ("external_name")  :readonly ;
      << variable declarations >>
```

The syntax for variable declarations is identical to that for the STATIC section. Note that if the optional ":readonly" attribute is present, each declaration must have an initializer (a constant expression) or HLA will complain.

HLA allows multiple declarations of the same segment. In this case, the additional declarations are simply concatenated to the end of the original segment. If multiple SEGMENT declarations of the same name appear in a compilation, the optional *external_name* field must be absent in all declarations or the string must be exactly the same in all declarations.

The symbol table fields for variables you declare in a segment are identical to those for STATIC objects except for the *owner* and *externalName* fields. The *owner* field will contain a pointer to the symbol table entry for the seg-

1. The term segment in HLA has very little to do with the concept of segments on the x86 CPU. A segment to HLA is just a block of memory where you may place related objects. Generally, segments are fully contain within their own memory management page or set of pages (i.e., data from two different segments never appears in the same memory page). Furthermore, on some OSes it is possible to control the access to pages (i.e., R/W vs. Read-only).

ment to which they belong (e.g., *segname* above). The *externalName* field will contain either the segment's name (if the optional external segment name string is not present) or the specified external name.

SEGMENT declarations are legal only at the global PROGRAM/UNIT level. You may not embed SEGMENT declarations within procedures, methods, iterators, namespaces, or classes. SEGMENT names are always external. If the optional *external_name* string is present, HLA will use this as the external name; if the *external_name* string is not present, HLA will use the *segname* identifier as the external name.

Note that HLA predeclares three segment symbol table entries for the STATIC, READONLY, and STORAGE segments. Other than the fact that these are predefined segments, they behave similarly to any other segment; the principle difference is that HLA understands the semantics of these segments a little better and can provide better diagnostic error messages (e.g., if you don't initialize a READONLY object or you attempt to initialize a STORAGE object).

Here's how SEGMENT identifiers you declare use the fields in a symbol table entry (keep in mind that this is a discussion of the SEGMENT identifier's entry, not the entries for the variables you declare in the SEGMENT):

lcname, truename

These symbol table fields contain the declared name of the *segment* identifier.

localSyms

This field contains a pointer to the root node of the sub-symbol table (binary search tree) for this segment.

seg

This field points at the last entry in the linear list of symbols in the sub-symbol table associated with this segment. HLA uses this entry to append new entries to the list.

segList

This pointer contains the address of the first entry in the linear sub-symbol table list associated with this segment.

owner

This field points at the object whose *localSymbols* field contains a pointer to the sub-symbol table that holds this segment's symbol.

lexLevel

Segment declarations may only occur at lex level one, so this field will contain one for user-defined segments (it may contain zero for HLA-defined segments like the *data*, *readonly*, and *bss* segments).

objectSize

This field has no meaning for a SEGMENT symbol.

pType

This field contains *Segment_pt*.

symClass

This field contains *Segment_ct* for SEGMENT identifiers.

isExternal

This field always contains true since segments are always external.

isForward

SEGMENT objects do not use this field. It always contains false for these objects.

isPrivate

This field does not apply to SEGMENT identifiers. Therefore, it always contains false.

pClass

SEGMENT identifiers are never parameters, so this field always contains *notp_pc*.

externName

This field contains a pointer to the SEGMENT S external name. If the external name was not explicitly provided, this will be the same name as the SEGMENT name. Note, however, that this string is physically different than the *trueName* field even if the strings are identical.

offset/address

This field has no meaning for segment declarations; it will contain zero.

Segment symbol table entries do not use any of the remaining fields in a symbol table entry.

enterSegmentID Function Prototype

The *enterSegmentID* function inserts a segment identifier into the symbol table if it does not already exist. If the segment identifier is already in the symbol table, then this function simply ensures that the external names match (if *externName* is not NULL) and then returns a pointer to the existing symbol table entry. Note that it is not an error to have multiple occurrences of the same segment in the source file; HLA simply adds the variables in the additional segments to the existing sub-symbol table for the segment.

```
procedure enterSegmentID
(
```

```
            symbol      :string;
            externName  :string
);    returns( "eax" );
```

The *symbol* parameter holds the segment's name, the *externName* holds the external segment name. If *externName* is NULL, then *enterSegment* will duplicate the symbol string and use this as the external name for the segment. The *enterSegment* function initializes *localSyms, seg,* and *segList* with NULL.

enterSegment Function Prototype

The *enterSegment* function adds a variable associated with a segment to the segment's sub-symbol table.

```
procedure enterSegment
(
            symbol      :string;
            externName  :string;
            pType       :pType_t;
            symType     :symNodePtr_t;
            theSeg      :symNodePtr_t;
      var   v           :value_t
);    returns( "eax" );
```

The *symbol* parameter contains the name of the variable to enter into the symbol table. The *externName* parameter points at the EXTERNAL name, if this symbol is an EXTERNAL object (the *symbol* and *externName* pointers must refer to different strings, even if the string data is identical). The *pType* and *symType* parameters specify the variable's type. The *theSeg* parameter points at the segment symbol table entry to which this variable belongs. The *v* parameter contains the initial data for this entry, if any.

NAMESPACE Symbol Table Entries

A NAMESPACE declaration encapsulates a declaration section and creates a sub-symbol table for those declarations. NAMESPACEs help prevent "namespace pollution" by requiring a prefix NAMESPACE identifier to refer to objects appearing in the NAMESPACE. The syntax for a NAMESPACE declaration is the following:

```
namespace nsID;

      << Namespace Declarations >>

end nsID;
```

NAMESPACE declarations may include CONST, VAL, TYPE, STATIC, READONLY, STORAGE, SEGMENT, PROCEDURE, ITERATOR, METHOD, and MACRO declarations. NAMESPACE declarations may not include VAR, NAMESPACE, PROGRAM, or UNIT declarations.

Here's how HLA uses the symbol table fields for NAMESPACE symbols:

lcname, truename

These symbol table fields contain the declared name of the namespace (*nsID* in the example above).

localSymbols

This field points at the root of the search tree for the local symbols within the NAMESPACE.

seg, segList

For NAMESPACE objects, these fields contain NULL.

owner

This field points at the object whose *localSymbols* field contains a pointer to the sub-symbol table that holds this symbol. This will always be the PROGRAM or UNIT identifier since namespace exist only at lex level one.

lexLevel

NAMESPACES are always global, hence this field will always contain one. Note that NAMESPACE declarations are legal within procedures, but the entry for the NAMESPACE symbol is still made at the global level.

objectSize

This field is meaningless for NAMESPACE declarations.

pType

The field contains the enumerated *pType_t* constant *Namespace_pt*.

symClass

This field contains *Namespace_ct*.

isExternal

NAMESPACES are never external, so this field contains false. Note that objects you declare within a NAMESPACE can be external; HLA does not support the *namespace.var* dot notation for EXTERNAL names. You have to specify a unique external name string to prevent external name space pollution if this is a problem.

isForward

This field has no meaning for NAMESPACE declarations and always contains false.

isPrivate

This field has no meaning for NAMESPACE declarations and always contains false.

pClass

This field is meaningless for NAMESPACE declarations and always contains *notp_pc*.

externName

Namespaces are always local. Hence this field always contains NULL.

offset/address

This field is meaningless for NAMESPACE declarations.

The remaining fields have no meaning in a NAMESPACE declaration symbol table entry.

enterNamespaceID Function Prototype

The enterNamespaceID function checks to see if the specified symbol is already present in the symbol table. If so, this procedure simply returns a pointer to the existing entry; if not, this procedure enters the symbol into the symbol table at lex level one (regardless of what lex level the declaration occurs at). Here is the prototype for this function:

```
procedure enterNamespaceID
(
            symbol      :string
);    returns( "eax" );
```

The *symbol* parameter holds the segment's name.

Note that there is no "*enterNamespace*" function. To enter a symbol into a NAMESPACE S sub-symbol table, you simply supply the address of the NAMESPACE S symbol table entry as the *owner* field in a typical *enterXXXX* call.

PROGRAM, UNIT, PROCEDURE, METHOD, and ITERATOR Symbol Table Entries

PROGRAMS, UNITS, PROCEDURES, METHODS, and ITERATORS share a common symbol table format. They all support a set of local variables; in addition, PROCEDURES, METHODS, and ITERATORS also allow parameters (for consistency, HLA's symbol table entries allow PROGRAMs and UNITs to have parameters even though none will ever appear). Here's how PROGRAM, UNIT, PROCEDURE, METHOD, and ITERATOR symbols use the symbol table fields:

lcname, truename

These symbol table fields contain the declared name of the PROGRAM, UNIT, PROCEDURE, ITERATOR, or METHOD.

localSyms

This field contains a pointer to the local variables for this procedure. Note that this pointer contains the address of the root node of the binary search tree for the sub-symbol table. Also note that any PROCEDURE, METHOD, or ITERATOR parameters also appear in this tree.

seg

This field points at the segment entry for the CODE (TEXT) segment. HLA creates this special segment object at lex level zero before the compilation begins.

segList

This field forms a linked list of code objects in the current compilation unit.

owner

This field contains a pointer to the symbol table whose scope contains the current identifier. For PROGRAMs and UNITs, this is the HLA lex level zero symbol table root (an anonymous symbol that doesn't have an identifier associated with it). For PROCEDURES, METHODS, and ITERATORS, this field points at the PROGRAM, UNIT, PROCEDURE, METHOD, or ITERATOR symbol to whose scope the current symbol belongs. See the discussion at the end of this section for more details.

lexLevel

This field holds the numeric lex level of this symbol.

objectSize

This field has no meaning for these types of symbols.

pType

The field contains the enumerated *pType_t* constant associated with this variable's data type. This is one of the following values (depending on the symbol type):

Proc_pt:	Object is a procedure symbol.
Method_pt:	Object is a method symbol.
ClassProc_pt:	Object is a procedure that is a class member.
ClassIter_pt:	Object is an iterator that belongs to a class.
Iterator_pt:	Object is an iterator symbol.
Program_pt:	Ob ject is a program or a unit symbol.

symClass

This field contains one of the following values, depending on the symbol's class:

Proc_ct:	Object is a procedure symbol.
Method_ct:	Object is a method symbol.
ClassProc_ct:	Object is a procedure that is a class member.
ClassIter_ct:	Object is an iterator that belongs to a class.
Iterator_ct:	Object is an iterator symbol.
Program_ct:	Ob ject is a program or a unit symbol.

isExternal

This field contains true if the symbol is (currently) declared as an external object. HLA sets this field to false when (if) it finds an actual procedure declaration later in the source file.

isForward

This field contains true if this is a forward declaration of a procedure. HLA sets this field to false when the symbol is actually defined.

isPrivate

This field contains true if this PROCEDURE, METHOD, or ITERATOR object appears in the private section of a class definition. This field is always false for PROGRAM and UNIT symbol table entries.

pClass

This symbol does not apply to these symbol types, so this field always contains *notp_pc*.

externName

If this is an external PROCEDURE, METHOD, or ITERATOR, then this field holds the external name of the object.

address

This field contains the address of the first statement associated with this object in HLA's intermediate code. For UNITs, this field contains NULL (since UNITs don't have any code associated with them).

returnsStr

This field points at an HLA string that the HLA compiler will substitute for a call to this object after HLA compiles the call to the object. HLA uses this string to implement "instruction composition" for PROCEDURE, ITERATOR, and METHOD invocations.

parms

This field points at the first element of a PROCEDURE, ITERATOR, or METHOD parameter list. Note that the *Next* field forms this linear list of parameters. The parameter list ends with either *Next* contains NULL or when the first symbol in this linear list has a *pClass* value of *notp_pc*.

lastLocal

This field points at the last entry in the linear list of symbols associated with a PROCEDURE, METHOD, ITERATOR, PROGRAM, or UNIT. HLA uses this pointer to add new symbols to the linear list in the sub-symbol table.

baseClass

For class procedures, class iterators, and methods, this field contains a pointer to the symbol table entry for the class to which this symbol belongs. For other objects, this field contains NULL.

parmSize

This field specifies the number of bytes of parameter data associated with this object. For PROGRAMs and UNITs this field contains zero.

callSeq

For procedures, iterators, and methods, this field contains one of the following *callSeq_t* values that specifies the call sequence/parameter passing sequence:

pascal_cs: Pascal calling sequence. Caller pushes parameters in left to right order and the procedure removes the parameters from the stack upon return.

stdcall_cs: Standard calling sequence. Caller pushes parameters in right to left order and the procedure removes the parameters from the stack upon return.

cdecl_cs: C calling sequence. Caller pushes parameters in right to left order and the caller removes the parameters from the stack upon return.

This field is meaningless for PROGRAM and UNIT objects.

hasFrame

This field contains true if HLA is to emit code to build the stack frame when compiling the procedure. It contains false if it is the programmer's responsibility to write this code.

hasDisplay

This field contains true if HLA is to reserve space for a display in the activation record. If *hasFrame* is also true, then HLA will emit the code to build the display as well. If *hasDisplay* is false, then HLA assumes that there is no display associated with this object.

alignStack

This field contains true if HLA is supposed to emit code to align the stack on a double word boundary after constructing the activation record. Note that *hasFrame* must also be true in order for HLA to emit this code.

enterProc Function Prototype

The following function inserts a PROCEDURE, ITERATOR, METHOD, PROGRAM, or UNIT object into the symbol table:

```
procedure enterProc
(
      symbol      :string;
      pType       :pType_t;
      owner       :symNodePtr_t;
      baseClass   :symNodePtr_t
);    returns( "eax" );
```

The *symbol* field is a string with the object's name. The *pType* field contains *Proc_pt, iterator_pt, ClassProc_pt, ClassIter_pt, Method_pt,* or *Program_pt* depending on the type of symbol to enter into the symbol table. The *enterProc* procedure automatically sets the *symClass* value based on the *pType* value. The *owner* parameter is the address of the symbol table entry whose sub-symbol table contains this object; for PROGRAM and UNIT objects, this points at the root node of the HLA symbol table (an anonymous symbol). If the current object is a class procedure, class iterator, or a class object, then *baseClass* contains a pointer to the symbol table entry for the class object to which the current object belongs. For other objects, this field should contain NULL.

The enterProc procedure initializes several fields in the symbol table entry. The fields it initializes are *left, right, next, lcname, trueName, localSyms, lastLocal, seg, segList, owner, lexLevel, objectSize (with zero), pType, symClass,* and *pClass*. It is the caller's responsibility to initialize other fields in this symbol table entry including *isExternal, isForward, isPrivate, externName, address, returnsStr, parms, parmSize, callSeq, hasFrame, hasDisplay,* and *alignStack*. The caller must initialize these fields because this information generally isn't known when first entering the symbol into the symbol table. Note that although *enterProc* initializes the *localSyms* and *lastLocal* fields, it does not process the parameters and local symbols for this object; *enterProc* simply enters a few symbols into the symbol table prior to the processing of the parameter list. It is the caller's responsibility to enter parameters and local symbols into the local symbol table.

The *enterProc* procedure actually enters the symbol twice in the symbol table. For the first entry, *enterProc* enters the symbol into the symbol table at the lex level correspoding to the PROCEDURE, METHOD, ITERATOR, PROGRAM, or UNIT's declaration. Then *enterProc* enters the symbol a second time into the symbol table; this second entry, however, is in the sub-symbol table associated with the first declaration. This entry is added to prevent (immediate) redefining the symbol as a local symbol in the PROCEDURE, METHOD, ITERATOR, PROGRAM, or UNIT, thus barring recursive invocations.

MACRO and TEMPLATE Symbol Table Entries