33 Tables Module (tables.hhf)

The HLA Tables module provides support for associative lookup tables. You can view a table as an array of objects that uses a string index rather than an integer index. The HLA table routines use a hash table to rapidly look up the specified string in the table and return a pointer to the specified element in the table.

Note: Because of their high-level nature, this document only provides high-level calling sequences for the table management procedures. Low-level calls are possible, but are generally so painful that they aren't worth making. If you are dead set on making low-level calls to *table* class methods and procedures, please consult the HLA documentation for directions on how this is done.

A Note About Thread Safety: The HLA standard library table module does not attempt to synchronize thread access to the table data structures. If you are going to be manipulating tables from multiple threads, it is your responsibility to ensure that the threads use properly synchronized access to this resource. This issue may be addressed in a future version of the standard library, for now it is your responsibility to ensure correct operation in a multi-threaded environment.

33.1 The Tables Module

To use the table class and methods in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "tables.hhf" )
```

or

```
#include( "stdlib.hhf" )
```

33.2 The Table Class

The HLA stdlib tables module consists of two classes: the *table* class and the *node* class. To create a table object, you first create a new *node* type by overloading the *node* class and then you can use the *table* class' methods to manipulate a table of these nodes.

To begin with, each element (called a *node*) in an HLA table uses the following structure:

```
tableNode_t:
    record

        link:     pointer to tableNode_t;
        Value:    dword;
        id:       string;

    endrecord;
```

The *link* field is for internal use by the table management routines; you should never modify its value.

The *id* field points at the string that indexes the current node in the table. You may use the value of this string, but you must never modify the characters in the string. The hashing function utilitized by the table management code will not be able to locate this string if you change the characters in the string after entering the string into the table.

The *Value* field is reserved for your own purposes. Other than initializing this field to zero when they create a table entry, the table management routines ignore this field. If you wish to associate data with an entry in the table you can either store the data directly in this field (if the data is four bytes or less), or you can allocate storage for the data outside the table entry and store a pointer to the data in this field (remember, pointers are four-byte objects).

The table_t data type is a class that provides the following methods and procedures:

```
procedure table_t.create( HashSize:uns32 );
method table_t.destroy( FreeValue:procedure );
method table_t.getNode( id:string );
method table_t.lookup( id:string );
iterator table_t.item();
```

Like most HLA classes, the table_t class provides a constructor named *table_t.create* and a destructor named *table_t.destroy*. The class also provides two additional methods and an iterator, *table_t.getNode, table_t.lookup*, and *table_t.item*. Although there doesn't appear to be much to this class, these few routines provide considerable power.

procedure table_t.create(HashSize:uns32);

The create procedure, being a static procedure constructor, is typically called in one of two fashions:

When dynamically allocating a table_t object on the heap and storing the pointer away into a variable whose type is "pointer to table_t":

```
table_t.create( some_constant );
mov( esi, PtrToTableVar );
```

When you've got a var or static variable object (named table_var_name in this example), you can use code like the following to construct the table_t object:

```
table_var_name.create( some_constant );
```

The table_t constructor requires a single uns32 parameter. This value should be approximately the number of entries (elements) you expect to insert into the table. This value does not have to be exact. Anytime you want to add a new element to the table, you may do so; there are no limitations (other than available memory) on the number of elements in a table. However, this parameter value is a hint to the table management routines so it can allocate a hash table of an appropriate size so that (1) access to the table elements is fast, and (2) the hash table doesn't waste an inordinate amount of space. If the hint value you supply is too small, the table lookup routines will still function properly, but they will run a little slower. If the hint value you supply is too large, the table management routines will waste some memory.

HLA high-level calling sequence example:

```
table_t.create( 128 );
mov( esi, someTableVarName );

SomeStaticTableVarName.create( 256 );
```

method table_t.destroy(FreeValue:procedure);

The *table_t.destroy* method frees up the data in the table. This routine deallocates the storage associated with the hash table, it deallocates the storage associated with each node in the table, and it deallocates the storage associated with each string (the *id* field in record *tableNode_t* above) in the table. Unfortunately, the *table_t.destroy* method doesn't know anything at all about the *Value* field of each node. If this is just some simple data, then the destructor probably doesn't need to do anything with the *Value* field. On the other hand, if *Value* is a pointer to some other data that was dynamically allocated, *destroy* should probably deallocate the storage associated with the *Value* field. Unfortunately, *destroy* has no apriori knowledge about the *Value* field, so it cannot determine if (or how) it should deallocate storage associated with *Value*.

To resolve the problem above, *table_t.destroy* calls a user-defined function that is responsible for cleaning up the data associated with *Value*. You will notice that *table_t.destroy* has a single parameter: *FreeValue*. This parameter must be the address of a procedure whose job is to handle the destruction of the *Value* field. If no clean up is necessary, you must still provide the address of some routine. That routine should simply return without further activity.

Upon entry into the *FreeValue* procedure, the EBX register contains a pointer to the current *tableNode_t* record being deallocated. At this point, none of the other fields have been modified; in particular, the *id* field is still pointing at the string associated with the node. The *FreeValue* procedure may access the *id* field, but it must not deallocate the storage associated with this string. The *table_t.destroy* method takes care of that after *FreeValue* returns.

The "tabledemo.hla" file accompanying the HLA release gives another example of how you could use the *FreeValue* procedure. This code doesn't deallocate any storage in this procedure (named *PrintIt* in this file), instead, it uses this call to dump the data associated with the node to the display when the table is freed.

```
HLA high-level calling sequence example:

  // Dummy free routine, nothing to do:

  procedure myFree; @noframe;
  begin myFree;
     ret();
  end myFree;

     .
     .
     .

  table_t.create( 128 );
  mov( esi, myTable );
     .
     .
     .
  myTable.destroy( &myFree );
```

method table_t.lookup(id:string); @returns("eax");

The *table_t.lookup* method locates a node in the table. The string parameter specifies which node to find in the table. On return, EAX contains the address of the corresponding *tableNode_t* record in the table, if the specified node is present (that is, the node's *id* field matches the string passed to *table_t.lookup*). If the node is not present in the table, then the *table_t.lookup* method returns NULL (zero) in the EAX register.

```
HLA high-level calling sequence example:

  table_t.create( 16 );
  mov( eax, tableVar );
     .
     .
     .
  tableVar.lookup( "someString" );
  if( eax <> NULL ) then

     stdout.put( "Found 'someString' in the table" nl );

  else

     stdout.put( "Did not find 'someString' in the table" nl );

  endif;
```

method table_t.getNode(id:string); @returns("eax");

The *table_t.getNode* method serves two purposes. First of all, it looks up the specified string value in the table. If it finds a node in the table corresponding to the string parameter, it returns a pointer to that (*table_t.tableNode_t*) node in the EAX register. If it does not find the string in the table, then it creates a new node and inserts the string into that new node; it also initializes the *Value* field to zero in this case. Note that *table_t.getNode* makes a copy (using *str.a_cpy*) of the string, it does not store the string pointer you pass directly into the id field. Upon return, EAX will point at the new node. Note that whether or not an existing node is present in the table, *table_t.getNode* will always return a pointer to the node associated with the specified string. It either returns a pointer to a pre-existing node or it returns the pointer to the new node.

If you want to insert a new node into the table and fail if the node already exists, you will need to first call *table_t.lookup* to see if the node previously exists (and fail if it does). You may then call *table_t.getNode* to insert a new node into the table. While it would be easy to add this functionality to the *table_t* class, it would be rarely used and probably isn't needed.

HLA high-level calling sequence example:

```
table_t.create( 16 );
mov( eax, tableVar );
tableVar.getNode( "someString" );// Insert "someString"
    .
    .
    .
tableVar.getNode( "someString" );// Retrieve ptr to "someString"
```

method table_t.getNode(id:string); @returns("eax");

The *table_t.item* iterator yields each node in the table during the execution of the corresponding foreach loop. Note that *table_t.item* does not yield the nodes in any particular (discernable) order. However, it will yield each item in the list exactly once. The iterator returns with EAX pointing at a *table_t.tableNode_t* object.

HLA high-level calling sequence example:

```
table_t.create( 16 );
mov( eax, tableVar );
    .
    .
    .
foreach tableVar.item() do

    // Process node pointed at by EAX.

endfor;
```