30 The HLA Standard Template Library

The following sections provide a basic description of some of the routines in the HLA Standard Template Library. Keep in mind that the HLA Standard Template Library is a work in progress and the following sections may not be totally up to date. The HLA Standard Template Library header file and source code is the final arbitor if there is a question how the routines operate.

Unless otherwise noted, you can assume that the Standard Library routines preserve all the general purpose registers. They generally do not preserve the flags.

30.1 Introduction to the HLA STL

The HLA Standard Template Library (STL) was designed to be similar to the C++ STL. The idea is not only to provide similar functionality to the C++ STL, but also to help make the transition from C++ to assembly language an easier process. Though the HLA STL is by no means an exact replicate of the C++ STL, the concepts are sufficiently close to allow someone to use the HLA STL in the same way they'd use the C++ STL without having to learn a new programming paradigm.

Though the HLA STL is especially easy to learrn by those who are familiar with templates in C++, it's also a relatively straight-forward package to learn by those who are not C++ programmers. The HLA STL package provides convenient code for declaring dynamic arrays, queues, lists, lookup tables, and other advanced data structures. By using HLA STL code, you'll find it much easier to write advanced assembly language code taking advantage of these sophisticated data structures.

"Template" is a special C++ term that is effectively a synonym for *macro*[1]. Therefore, one big difference you'll find between the HLA STL and the HLA Standard Library is that there are not object files you link in with code that uses the HLA STL. The STL is simply a set of macros that you incorporate into your program by including the "stl.hhf" header file and then invoking the templates (macros) that interest you. Therefore, to use the HLA STL package, the first thing you must do is include the following statement in your HLA program:

#include("stl.hhf")

Note that HLA "stdlib.hhf" header file does not automatically include the STL header file. The STL and the HLA Standard Library are two separate packages and you must explicitly include "stl.hhf" to use the HLA STL facilities.

The HLA STL is a set of macros (templates) that create user-defined class objects when you invoke them. To a programmer, these macros look somewhat like user-defined types that you use in a type declaration section. For example, consider the following *vector* type declaration:

```
type
  int32Vector :stl.vector( int32 );
```

The principle difference between an STL type declaration and a standard type declaration is the fact that STL declarations are *parameterized*. STL types are abstract data types that usually *contain* some other type. A vector type, for example, is a dynamic array type, with each element of the vector being some base type (*int32* in the vector example above). It is possible to have vectors of 32-bit integers (int32), characters, strings, or any other built-in or user-defined data type. The parameter associated with an STL declaration specifies the underlying data type on which the new type is built. Consider the following two vector declarations:

```
type
  int32Vector :stl.vector( int32 );
  stringVector :stl.vector( string );
```

These two declarations create two new class types, an int32 vector and a string vector, that one can use to declare integer and string vectors. It's inportant to realize that the vector template creates different types, not variables. It's also important to realize that vector types are different. That is, int32Vector and stringVector, although both vectors, are not compatible types.

1. Technically, this is not true, but we'll ignore the distinction in this document.

30.2 Type Declarations Created by a Template

Templates only create types, not variables. In order to create actual variable objects, you must declare such objects in an HLA *var, static*, or *storage* section (because all template types are classes, you cannot create initialized class objects in a readonly [or static] section).

Template expansions may only occur in an HLA *type* section at the global level of a program or unit. This is because the template expansion, in addition to creating the data type, emits the code for all the class methods and the VMT for the class. After a template expansion, the template will leave the program in the *type* declaration section, but keep in mind that internally, the template expands to code and data in addition to type declarations.

In addition to the user-specified type name, an STL declaration typically creates two or three other types during expansion. Most templates will also create the following types:

```
type
  p_name :pointer to name;
  name_cursor: pointer to XXXX;
```

where *name* is the user-specified type name (e.g., int32Vector and stringVector from the previous examples, yielding int32Vector_cursor and p_int32Vector). *XXXX* represents an unimportant type name for our purposes; Cursor types are opaque insofar as an HLA application will use a cursor type to pass data amongst template class methods without needing to know what the type actually references.

Some template types (e.g., *list* and *table*) also create a node type, declared as follows:

```
type
  name_node: record
        data:parameter_type;
        <<other_fields>>
  endrecord;
```

where *parameter_type* is the type passed to the STL template as a parameter (e.g., *int32* and *string* in the current examples) and *other_fields* represent some opaque fields that the class' methods reference and should be treated as private data by all other code.

Once you invoke a template in a type declaration section, you can create actual objects using the template's resulting type. For example, to create *int32Vector* and *stringVector* objects, you could use declarations like the following:

```
type
  myInt32v :int32Vector;
  myStringv :stringVector;
```

Note that you do not use an STL template when you define an actual variable. You use templates to create types and then you use those types you've created to declare variables.

30.3 Template Objects are Classes

Though it is not particularly apparent from the invocation of a template, you should realize that HLA STL templates create class types. When you declare a variable of some template type you've defined, you're creating a class object. Therefore, it helds if you're familiar with the HLA object-oriented programming paradigm (and, in particular, HLA classes, methods, class procedures, and class iterators).

Note that when you define a new type using an HLA template, that type definition also creates a set of methods, procedures, and iterators specific to that class. That is, a type declaration like the following:

```
type
  csetVector :stl.vector( cset );
```

does a lot more than simply define a type – it also expands to a lot of code to your source file. A typical template class may have 2-4 dozen methods, class procedures, and class iterators associated with the class. Each time you create a new class by expanding an STL template, you get a new copy of all those routines. Consider the following pair of type declarations:

```
type
```

```
int32Vector :stl.vector( int32 );
i32Vector :stl.vector( int32 );
```

Internally, these classes are exactly the same. Externally, however, they are different types. Therefore, the HLA STL will create a separate set of methods that are *absolutely identical* in everything but name for these two types. This is a waste of space. In general, you should only create one instance of a particular STL class, so that you only expand one set of methods, procedures, and iterators (and a virtual method table [VMT]) for that one class. If you really two different "vector of int32" types, you should consider a declaration like the following:

```
type
  i32v :stl.vector( int32 );
  int32Vector :i32v;
  i32Vector :i32v;
```

HLA only generates one set of methods/procedures/iterators and VMTs for the i32v class; the int32Vector and i32Vector classes share this code and VMT (which is reasonable, as the operations on *int32Vector* and *i32Vector* types will always be the same).

Note that the HLA STL macros emit *different* methods, procedures, and iterators for class objects with differing underlying types. For example, the *int32Vector* and *stringVector* types both need their own set of specific methods/procedures/iterators because those routines operate on completely different data types.

So keep in mind that everytime you expand an HLA STL template, you get a new set of routines associated with the corresponding class you've defined. Of course, you only have one set of routines for each class you create, regardless of how many instances (class variables) of that class you define. That is, declaring multiple variables does not cause the emission of multiple sets of methods; only defining *types* will do this.

30.4 Class Traits

A template trait is a compile-time or run-time value that provides some information about the type of the underlying class. The HLA STL defines several common trait objects that are testable within any STL type. Advanced programmers may use conditional assembly to test compile-time traits or actual machine instructions to test run-time traits. By utilizing traits, your code can behave differently, depending on the underlying (template) data type.

Though most programmers can use STL class types without ever worrying about traits, the availability of traits makes it possible to do some very sophisticated things with the HLA STL. Where traits might come in handy is when you're writing your own macros to which you pass different STL objects and you might need to generate different code (or emit an error message) depending on the traits the object supports. Also, you soon find out that it's possible to create an object with fewer traits than the default object supports. For example, if you're using some simple int32 *vector* types and you don't require any of the cursor capabilities, you can tell the HLA STL to construct an int32 *vector* type without cursor support (thus reducing the amount of code the template generates). However, if you pass one of these vector objects to a generic macro that works with vectors, the lack of cursor support could create a problem. Fortunately, traits solve this problem by letting that macro (or even some run-time code) test to see whether cursor support is present, generating an error (or otherwise handling the situation) if cursors are not available.

30.4.1 isSTL_c Trait

The most fundamental trait assocated with all template classes is the *isSTL_c* trait. This is a compile-time constant (*const* class declaration) that is defined and set to TRUE for all STL types. You can use the HLA compile-time *@defined* function to test whether or not the *isSTL_c* field is defined for a given class type. If this symbol is defined, then you can generally assume that the underlying class is an HLA STL class and you can test for any of the other STL traits[2]. Here's how you'd typically use this trait:

```
#if( @defined( someType.isSTL_c ))
  << code to compile, knowing that this is an STL type >>
#endif
```

2. Of course, there is nothing stopping someone from defining this constant in some arbitrary non-STL class, but you can generally assume that someone won't hijack your program's logic by doing this.

30.4.2 Compile-Time Traits

If the *isSTL_c* field is defined, then the class will also define three dword constants: *hierarchy_c*, *capabilities_c*, and *performance_c*. These constants are bit maps with each (defined) bit posititon corresponding to some capability, or lack thereof, of the current class object. If the bit position contains one, then the class possesses the corresponding capability; if the bit position contains zero, then the class lacks that capability.

The hierarchical traits specify which subclasses are associated with a given type. The capability traits specify which general methods are available to a given class. The performance traits provide an indication of the performance of the methods available to a given class.

The STL defines the following constants (which are all values with a single bit set):

Hierarchical traits (testable in *hierarchy_c*):

- stl.isContainer_c
- stl.isRandomAccess_c
- stl.IsArray_c
- stl.isVector_c
- stl.isDeque_c
- stl.isList_c
- stl.isTable_c

Capability traits (testable in *capabilities_c*):

- stl.supportsOutput_c
- stl.supportsCompare_c
- stl.supportsInsert_c
- stl.supportsRemove_c
- stl.supportsAppend_c
- stl.supportsPrepend_c
- stl.supportsSwap_c
- stl.supportsForEach_c
- stl.supportsrForEach_c
- stl.supportsCursor_c
- stl.supportsSearch_c
- stl.supportsElementSwap_c
- stl.supportsObjSwap_c
- stl.elementsAreObjects_c

Performance traits (testable in *performance_c*):

- stl.fastInsert_c
- stl.fastRemove_c
- stl.fastAppend_c
- stl.fastPrepend_c
- stl.fastSwap_c
- stl.fastSearch_c
- stl.fastElementSwap_c

As their category suggests, you use these constants to test particular bits in the *hierarchy_c, capability_c*, and *performance_c* compile-time variables, respectively. For example, if you have an STL class and you want to determine if it is a *vector* class, you could use code like the following:

```
#if( (mySTLObject.hierarchy_c & stl.isVector_c) <> 0 )

  << you can assume mySTLObject is a vector object here >>

#endif
```

30.4.3 Run-Time Traits

If the *isSTL_c* field is defined, then the class object provides three run-time dword variables containing various set bits that determine the characteristics of that class. These run-time traits are the same as the compile-time traits except, of course, you can test their values at run-time using machine instructions. These variables are

hierarchy, capabilities, and *performance*. They are run-time analogs to the compile-time constants mentioned in the previous section and are associated with the same set of trait constant, i.e.,

Hierarchical traits (testable in *hierarchy_c*):

- stl.isContainer_c
- stl.isRandomAccess_c
- stl.IsArray_c
- stl.isVector_c
- stl.isDeque_c
- stl.isList_c
- stl.isTable_c

Capability traits (testable in *capabilities_c*):

- stl.supportsOutput_c
- stl.supportsCompare_c
- stl.supportsInsert_c
- stl.supportsRemove_c
- stl.supportsAppend_c
- stl.supportsPrepend_c
- stl.supportsSwap_c
- stl.supportsForEach_c
- stl.supportsrForEach_c
- stl.supportsCursor_c
- stl.supportsSearch_c
- stl.supportsElementSwap_c
- stl.supportsObjSwap_c
- stl.elementsAreObjects_c

Performance traits (testable in *performance_c*):

- stl.fastInsert_c
- stl.fastRemove_c
- stl.fastAppend_c
- stl.fastPrepend_c
- stl.fastSwap_c
- stl.fastSearch_c
- stl.fastElementSwap_c

Because these are run-time values, you must you 80x86 machine instructions to test for these trait value, e.g.,

```
test( stl.supportsCursor, mySTLObj.capabilities );
if( @nz ) then

  << execute code that uses the cursor methods in the object >>

endif;
```

30.4.4 Trait Constants

The following subsections define the meaning of each of the traits. Note that the term "true" means that the trait value is non-zero (and will have a single set bit, the bit position determined by the particular triat) while "false" means that the trait's value is zero.

30.4.4.1 stl.isContainer_c Trait

This bit is set in *heirarchy_c* or *hierarchy* if the STL object is (known to be) a container object. Currently, almost all STL objects are containers so this trait will be true. (The only STL object that is not a container is the base object, and you'll generally not declare any STL objects using the base type).

A container is a type that holds elements of some other type. All common STL types are container types as they are all composite data types (e.g., arrays, lists, tables, and so on).

30.4.4.2 stl.isRandomAccess_c Trait

This bit is set in *heirarchy_c* or *hierarchy* if the underlying STL object provides efficient (O(1) time) random access to the underlying type's elements. Vectors and deques are examples of objects that support random access as it takes the same amount of time to access any arbitrary element of these types.

30.4.4.3 stl.isArray_c Trait

This bit is set in *heirarchy_c* or *hierarchy* if the underlying class is an array object. Currently, this value is true for *vector* and *deque* types.

30.4.4.4 stl.isVector_c Trait

This bit is set in *heirarchy_c* or *hierarchy* if the underlying class is a *vector* class type.

30.4.4.5 stl.isDeque_c Trait

This bit is set in *heirarchy_c* or *hierarchy* if the underlying class is a *deque* class type.

30.4.4.6 stl.isList_c Trait

This bit is set in *heirarchy_c* or *hierarchy* if the underlying class is a *list* class type.

30.4.4.7 stl.isTable_c Trait

This bit is set in *heirarchy_c* or *hierarchy* if the underlying class is a *table* class type.

30.4.4.8 stl.supportsOutput_c Trait

This bit is set in *capabilities_c* or *capabilities* if the underlying class supports a *toString* method that the HLA Standard Library can employ in macro invocations such as *stdout.put* or *fileio.put* to write the class' data to some output stream. By default, this value is false. If you provide a *toString* method for a given data type you define, then you'll set this constant to true.

30.4.4.9 stl.supportsCompare_c Trait

This bit is set in *capabilities_c* or *capabilities* if the underlying class supports the *isEqual, isLess,* and *isLessEqual* methods. Some classes may only support an *isEqual* method, in which case the *supportsCompare_c* trait will be false; you may test for *isEqual* by using the *@defined* compile-time function.

30.4.4.10 stl.supportsInsert_c Trait

This bit is set in *capabilities_c* or *capabilities* if the class supports data insertion into an object. Generally, this implies that you have at least an *insertVal* and an *insertRef* method available. Other insertion methods may also be available, use *@defined* to test for their presence if you need to determine whether they exist for a particular class object. Not all class types accept the same parameter lists for their insert methods, thus limiting the generic usefulness of these methods (e.g., *table* insertions are based on strings rather than an integer index). You can test the *is****_c* traits (e.g., *isTable_c*) to handle such cases.

30.4.4.11 stl.supportsRemove_c Trait

This bit is set in *capabilities_c* or *capabilities*e if it is possible to remove objects from an STL object at run-time. Some STL data types (e.g., tables) do not allow the removal of an object once it's been inserted into the object; such types will (obviously) set *supportsRemove_c* to false.

30.4.4.12 stl.supportsAppend_c Trait

This bit is set in *capabilities_c* or *capabilities* if it iis possible to append a data element to the end of some STL object in memory. Some STL data types (e.g., tables) do not support the notion of a data sequence and, therefore, do not define an append operation. You can test this compile-time constant to check whether appending is a valid object before attempting it.

30.4.4.13 stl.supportsPrepend_c Trait

This bit is set in *capabilities_c* or *capabilities* if it iis possible to insert a data element at the front of some STL object in memory. Some STL data types (e.g., tables) do not support the notion of a data sequence and,

therefore, do not define a prepend operation. You can test this compile-time constant to check whether appending is a valid object before attempting it.

30.4.4.14 stl.supportsForEach_c and supportsrForeach_c Traits

These two compile-time constants (testable in *capabilities_c* and *capabilities*) tell you whether the template type supports a forward iterator (ForEachElement) or a reverse iterator (rForEach). Most STL types, by default, provide both types of iterators, even when the underlying data type is not a sequence (e.g., tables). For those data types that do not enforce an underlying sequence, the iterators will sequence through each of the object's elements, but the order of the sequence is undefined.

30.4.4.15 stl.supportsCursor_c Trait

Cursors are special opaque pointer objects that STL methods use to provide access to the underlying objects of some STL type. If an STL type supports cursors and operations on the type via those cursors, then this trait will be true for that type. This bit is set in *capabilities_c* or *capabilities* if the class supports cursors.

30.4.4.16 stl.supportsSearch_c Trait

(to be defined; unused as this is being written.)

30.4.4.17 stl.supportsElementSwap_c Trait

This bit is set in *capabilities_c* or *capabilities* if there is a *swapElements* method for the underlying class type. This method physically swaps the data between two elements of the STL type. This operation is not permitted for certain data types (e.g., tables), in which case the method will not exist and this trait will contain false.

30.4.4.18 stl.supportsObjSwap_c Trait

This bit is set in *capabilities_c* or *capabilities* if the *swapObj* method is present. *swapObj* will completely swap the values of two STL variables (whole objects, not elements of those objects).

30.4.4.19 stl.elementsAreObjects_c Trait

This bit is set in *capabilities_c* or *capabilities* if the elements of a given STL type are themselves class objects. This trait is set to false if the underlying data type is something other than a class. You may test this constant, for example, to determine if you should call constructors or destructors for each object created for an STL container class.

30.4.4.20 stl.fastInsert_c Trait

This bit is set in *performance_c* or *performance* if the class supports insertion and insertion can be (typically) done in O(1) time. It will be set to false if the class does not support insertion or executes slowly (generally O(lg n), O(n), or worse, time is considered "not fast").

30.4.4.21 stl.fastRemove_c Trait

This bit is set in *performance_c* or *performance* if the class supports element removal and removal can be (typically) done in O(1) time. It will be set to false if the class does not support removal or executes slowly (generally O(lg n), O(n), or worse, time is considered "not fast").

30.4.4.22 stl.fastAppend_c Trait

This bit is set in *performance_c* or *performance* if the class supports element append and append can be (typically) done in O(1) time. It will be set to false if the class does not support append or executes slowly (generally O(lg n), O(n), or worse, time is considered "not fast").

30.4.4.23 stl.fastPrepend_c Trait

This bit is set in *performance_c* or *performance* if the class supports element prepend and prepend can be (typically) done in O(1) time. It will be set to false if the class does not support prepend or it executes slowly (generally O(lg n), O(n), or worse, time is considered "not fast").

30.4.4.24 stl.fastSwap_c Trait

This bit is set in *performance_c* or *performance* if the class supports whole object swap and swapping can be (typically) done in O(1) time. It will be set to false if the class does not support object swap or it executes slowly (generally O(lg n), O(n), or worse, time is considered "not fast").

30.4.4.25 stl.fastSearch_c Trait

(as this was being written, this trait was undefined.)

30.4.4.26 stl.fastElementSwap_c Trait

This bit is set in *performance_c* or *performance* if the class supports element swap and swapping can be (typically) done in O(1) time. It will be set to false if the class does not support element swap or it executes slowly (generally O(lg n), O(n), or worse, time is considered "not fast").

30.4.5 Other Run-Time Traits

All objects possess two run-time fields: *typeName* and *isAlloc*. The typeName field is a string that provides the actual name of the object. For example, given a declaration like the following:

```
type
  int32Vector :stl.vector( int32 );
```

then the *typeName* field will be initialized with the string "int32Vector". The *typeName* variable should be treated as a read-only object; you should not modify the pointer or the string data associated with it (actually, the string data is in a read-only section, so you cannot modify it, but you should modify the string pointer, either).

The *isAlloc* field will contain true if the object has been allocated on the heap, it will contain false if this object was not allocated on the heap. The destructor method (destroy) uses this field to determine whether it needs to deallocate storage when the object is deleted. You may read the value of this field, but you must not change it.

Container objects (which is all STL objects at this point) have two additional fields: *numElements* and *containerName*. The *numElements* field is a un32 variable that specifies the number of objects contained within the container (e.g., the number of *vector* elements or *list* nodes). You must not modify this field; treat it as a read-only object; indeed, there is a *getSize* method that you can use to retrieve the value of this field. You should use the *getSize* method and avoid accessing this field directly.

The *containerName* field is a string that specifies the container type. This will typically be a string like "vector", "deque", "list", or "table". You should treat this as a read-only field.

The *arrayContainer, vector, deque, list,* and *table* classes all contain their own private data fields. You should treat all these fields as opaque – that is, private to the class – and you should not modify or even read their values. Where necessary, these classes will provide accessor functions that return the values of these data fields.

30.5 The Vector Template

(to be written)

30.6 The Deque Template

(to be written)

30.7 The List Template

(to be written)

30.8 The Table Template