C++
Symmetric List Processor
(Gnu SLIP)
Users Manual

Table of Contents

Tables

Examples

Figures

1 .0 Overview

In this document the name SLIP refers to the C++ Gnu SLIP implementation unless context or terminology dictate otherwise.

SLIP is a suitable replacement for the C++ Standard Template Library (STL) List container, <list> or the Queue container <queue>, template when the memory usage in the container is unsuitable for the application or the container's functionality is too limited. The STL List container should be the API of choice otherwise.

SLIP is an API which supports manipulation of complex graphs, trees, and lists. It provides the capability to iterate over a structure and to create and insert lists or other structures, aka sublists, within a list in order to create graphs and trees. It uses its own space management utilities and uses the heap retrieve large globs of memory, does not have garbage collection, and is developed with performance as an issue. Its goal is to provide extensive graph-centric capability and to be suitable in both embedded and desktop systems. It is fit for use.

Where possible, the method names and method semantics conform to normal software development and C++, vocabularies. Operations available should be familiar to C++ developers and Software Engineers.

Primitives required to manipulate lists, delete and move lists and cells, insert new cells into lists and other required list operations are included. List referencing ability is included and allows multiple list references to the same list. The set of operations is meant to be comprehensive and extensible, allowing application driven extensions of basic capabilities.

Using cells as a fundamental data type allows the user to write expressions such as X = Y, X == Y, X += Y and so on, where X and/or Y can be a SLIP cell or a C++ literal or variable. Computations with SLIP cells yield the same results on 32/64-bit architectures.

Within the API framework an application can create its own data types dynamically. These application defined data types are first class types which can participate in all operations, including assignment, arithmetic, bit, and logical, as well as those specific to list manipulation.

No fuss, no muss, and no bother. The API is:

- Performance-centric. Attention is paid to short call sequences.
- Space-centric: SLIP dynamic space is defined by the user and there is no garbage collection.
- Application-centric: SLIP data types are treated as primitive C++ types.

- Lists, trees, and graphs are supported.

Is it good? Depends on who 'you' are. It's liability is that with the richness in vocabulary there is a stiff learning curve.

1.1 History

The Symmetric List Processor (SLIP) (1)(3) was developed by Dr. J. Weizenbaum in 1963 as an application programming interface (API) in FORTRAN II. SLIP can support acyclic graphs, trees and lists but it can not correctly delete cyclic graphs (recursive lists). Previous work by Weizenbaum(2) lead him to create SLIP. SLIP is a Symmetric List Processor in the sense that links are bi-directional, allowing list traversal in a 'forward' or 'backwards' direction with equal facility.

The FORTRAN II SLIP(3) code was used as a vehicle for a number of implementations on different computers(5)(6)(11)(12) with generally good results, SLIP, performing the role as a symbolic processing language, was used in non-mathematical implementations as a transactional psychoanalyst, ELIZA(13), to perform symbolic solutions to systems of ordinary differential equations(14), and to perform symbolic list operations(17).

One early issue was the inability of SLIP to recover cells from a cyclic graph(4). This issue was addressed by Weizenbaum(6) as seldom occurring and needing little attention. Recovery of space for cyclic graphs remains a pressing issue. Proposals to use garbage collection (mark and sweep algorithms) were variously proposed as a solution(15)(19). SLIP is an API(8). Weizenbaum proposed a solution to recovering the space in a cyclic graph by requiring the user to provide information on when a list is being used(16) and then performing a garbage collection sweep when space is exhausted by recognizing when a list is in active use, and can not be recovered, from when a list is not in active use, and can be recovered.

SLIP uses constant sized cells. This aids in memory recapture but inhibits the generalized notion of a list processor able to use any sized cell for any operation. A paper which addressed using variable sized cells and the elaboration of garbage collection to accommodate this was proposed(18). Other than the authors work there apparently was no traction to the concept and no other documented efforts. This version of SLIP provides a means for an application to create variable sized data items within a standard SLIP cell, but requires that data creation and deletion operations be supported by the application.

1.2 Features

SLIP data cells are heterogeneous (they can changed dynamically, and they can be any one of C++ types bool, char, unsigned char, long, unsigned long, string or double, and SLIP types sublist reference (**SlipSublist**) or a user defined data type.

SLIP data cells can participate in operations as if they were C++ variables. That is, if X is a SLIP data cell, and Y is a Slip data cell or a C++ literal or variable, then all the below operations are legal:

X op Y	Where op = {+ - * / % << >> | & < <= == => > }
Y op X	Casting is done as required and in conformance to C++
X op= Y	Where 'op=' = { = += -= *= /= %= <<= >>= |= &= ^= }
Y op= X	Casting is done as required and in conformance to C++
(cast)Y	A Slip cell can be cast in any one of the defined C++ types according the C++ casting rules.
op Y	Unary operations are supported {! ~ + -}.

The legality of the operation depends on the semantics of the data type of the SLIP cell.

An application can create a data type and SLIP will manage it. The application data type can be configured to take part in C++ operations in a way determined by the application. All of the C++ unary, binary, and casting operations are available for redefinition.

Input / Output is round-trip. Input of a graph will recreate the original graph and data exactly. This includes the graph topology and all SLIP data (bool, char, unsigned char, long, unsigned long, string, double, and user data types). This allows retention of data between applications and/or storage of intermediate values in ASCII.

Input and Output is ASCII. The user can create a graph in an editor for input into a SLIP defined application. This allows the creation of configuration files, the construction of partial results, and the incorporation of externally created acyclic graphs.

If () is a list, then so is (()). This allows construction of trees and graphs. For example, a 3 deep binary tree can be represented as: ((()) (())). There is no inherent limit to the depth of trees or graphs.

Lists can be shared. A list created for one purpose can be shared in other lists.

Lists can be act as a stack, a queue, or a list. Primitives are provided for stack operations (pop, push), for queue operations (enqueue, dequeue), and for iteration. SLIP does not impose a use and a list can be used in all modes during an application.

List operations are time bounded:

1. Movement: O(1).
2. Delete: O(1).
3. Flush: O(1).
4. Copy: O(n).
5. Insertion: O(1).

Lists can have an associative list (called a Descriptor List). Each list can have an associative list containing <key value> pairs, with operations to return a value based on a key.

Iterators are available for list traversal. The iterators can optimally traverse a list and sublists, or can traverse a list and return from sublists. The later facility allows non-recursive entry into complex graphs.

Memory is managed by the SLIP systems. There is no fragmentation and there is no garbage collector.

Memory size used for SLIP cells is not limited and can grow dynamically. If the application requires more SLIP cells than initially allocated the allocation amount will increase dynamically.

1.3 SLIP vs STL List Containers

This comparison is fair. The intent is not to promote SLIP but to allow software engineers a standard for judging suitability for use in a given application.

Most SLIP functionality can be hosted onto STL Lists. But each such effort comprises some labor. The table presents SLIP functionality in comparison to basic STL List functionality without considering any transformations of the STL List to yield the same results as SLIP. STL is considered as it is and not as it might be.

Table 1.3-1: SLIP vs STL List / Queue Containers

Criteria	SLIP	STL	**Description**
Learning Curve	long	short	Time to learn methods and use
Memory fragmentation	no	yes	Can memory be fragmented over time
Embedded system use	yes	no	Use of heap in STL precludes use in embedded systems
I/O	yes[2]	no	STL provides no I/O functionality
Graphs/Trees	yes	no	STL supports lists, SLIP supports graphs
List sharing	yes	no	STL supports lists and does not support embedded lists
Utility functions	no	yes	STL provides sort, remove_if, etc.
Heterogeneous data	yes	no	STL is not designed for dynamically changing data types
C++ operations native	yes	no	STL has no mechanism for data to participate in op's

[1] SLIP allows creation of SLIP data cells in a method and allows non-anonymous lists to be created. If they are not deleted a hole will be created in SLIP managed space.
[2] SLIP I/O should not be used in embedded systems. Output uses recursion and the stack, input is iterative and uses the heap.

Although SLIP has more functionality than STL containers, that functionality comes at a cost in learning, discipline in using, and an increased memory footprint. If that functionality is not needed then it is better to use the STL containers, or if the project is simple enough, to build your own.

1.4 Theory of Operation

Include “slip.h” in each file where SLIP operations are being performed.

Before any SLIP operation is done, a slipInit should be executed. This acquires memory from the heap and initializes it. If slipInit is not executed, then at the first SLIP operation requiring memory, SLIP uses default values.

After all SLIP operations are completed, a deleteSlip should be executed. This returns all memory to the heap. In order to continue after a deleteSlip, the application must execute a slipInit.

There is no concept of a SLIP object and there is no need to support a global object or function parameter containing such an object. Once SLIP is initialized, SLIP can be used.

```
# include slip.h
main() {
  SlipCellBase::slipInit(INITIALALLOCATION, DELTAALLOCATION);
  … your code …
  SlipCellBase.deleteSlip();
}
```

or

```
# include slip.h
main() {
  … your code …
  SlipCellBase.deleteSlip();
}
```

and in each file using a SLIP method:

```
# include slip.h
func() {
  … your code …
}
```

1.5 C++ Implementation

This is a re-imagining of SLIP using C++. The basic structure and architecture is retained and the changes are in keeping with current method operations and names. The original SLIP is expressive and complete, and this is unchanged. The superficial differences are more a factor of a new language and removal of archaic features then a matter of substance. In broad terms, the organization of this implementation is the same as the original, FORTRAN II program.

FORTRAN II/IV/66/77 have a 6 character restriction on names. This has been removed and name changes have been made to make underlying functionality clearer. These changes do not change the underlying method semantics. In addition, names have been supplied to methods for concepts in common use which were not common during the initial implementation. These names, such as **push()** and **pop()**, use existing functionality, only giving a different name with known and common semantics.

Overview

SLIP is a symmetric list API. Each SLIP cell contains a pointer to the preceding and succeeding cell, hence the list is “symmetric”. The **SlipHeader** cell is the list header and is a unique cell with special properties. The preceding cell to the **SlipHeader** is the list tail (last cell in the list) and the succeeding cell is the list top (the first cell in the list). For a null list, the list top and tail is the **SlipHeader**. For a list with a single item, the list top and tail are the same and this is the item.

A prime concern of the implementation has been to provide an API “friendly” to embedded applications, both in terms of space utilization and in performance. The performance related issues are not interface concerns and do not alter the interface or method functionality. The specifics of the changes are contained in the underlying code and are presented in the SLIP Reference Manual. The general nature of the optimization is to reduce the calling depth to implement a given method, and to increase the use of inline code and/or to duplicate code as required. Note that inline code is not available in the produced DLL's. To get the full functionality and speed of inline code, SLIP must be compiled with the application.

This implementation (like the original) uses internal space management to control space usage. The implementation focus is to provide required deterministic and space conserving functionality for embedded systems and at the same time not limit usage to embedded frameworks. This notion is built on the concept of using fixed size SLIP cells, allowing uniform and straight-forward memory management, and provided a non-limited, dynamic expansion of this space as required. The space grows dynamically.

Lists must be acyclic graphs. A cyclic graph is not recoverable to the free space list, called the available space list (AVSL). With no loss in generality, acyclic graphs, trees, and (simple) lists are all be handled by SLIP in a uniform and consistent manner.

Archaic features, such as saving the current state in preparation for recursion, have been removed. The current implementation language (C++) contains the required mechanisms to support recursion and other features in C++ make some features redundant or intrusive or unnecessary.

Other features have been removed both to enable good programming practices and to allow the application to focus on the problem rather than list handling mechanics. Prime amongst these features is the removal of all requirements for the application to construct, remove, or change cell link pointers. The provided API performs these operations and the application is prohibited from doing them.

One augmentation not found in the original implementation is the use of datum. The original SLIP was not aware of the type of datum in a SLIP cell. The user was required to extract the datum and use application specific knowledge to infer the type and perform operations on the extracted data. The current implementation is aware of the type of datum stored into a cell, and will perform data type operations based on the known type. A SLIP cell containing data is considered as an atomic type with respect to usage. That is, without extraction of the datum in a SLIP cell, the datum is allowed to participate in logical, bit and arithmetic operations. Simply, $cell_1 + cell_2$ makes sense.

The application is allowed to put data into a SLIP data cell of the applications devising. The mechanism separates the application management of data and operations from SLIP and allows the application to provide data of any type and any size, subject to application provided space management. That is, space management is an application and not a SLIP responsibility. In a similar way, the application can override the SLIP provided operations to allow the application data to participate in operations, such as $cell_1 + cell_2$, or the application can provide any convenient class mechanism to augment the standard operations. If the election is to override existing SLIP operations, then the SLIP kernel will perform operations, such as $cell_1 + cell_2$, on the applications behalf. Otherwise in a manner analogous to the original SLIP API, the application must extract the data and/provide specific application defined data operations.

The implementation uses three Slip Cell classes, SlipHeader, SlipSublist, and SlipDatum, and two iterators, the SlipReader and SlipSequencer. The SlipHeader distinguishes a list header, the SlipSublist is a reference to a list, and the SlipDatum contains application specified data. The SlipReader is a structural reader which contains a memory of entered sublists, and the SlipSequencer is a structural reader which does not contain a memory of entered sublists.

The representation of “**(1 2 (3 4) (5))**” is given in Figure 1.5-1 Functional List Representation.

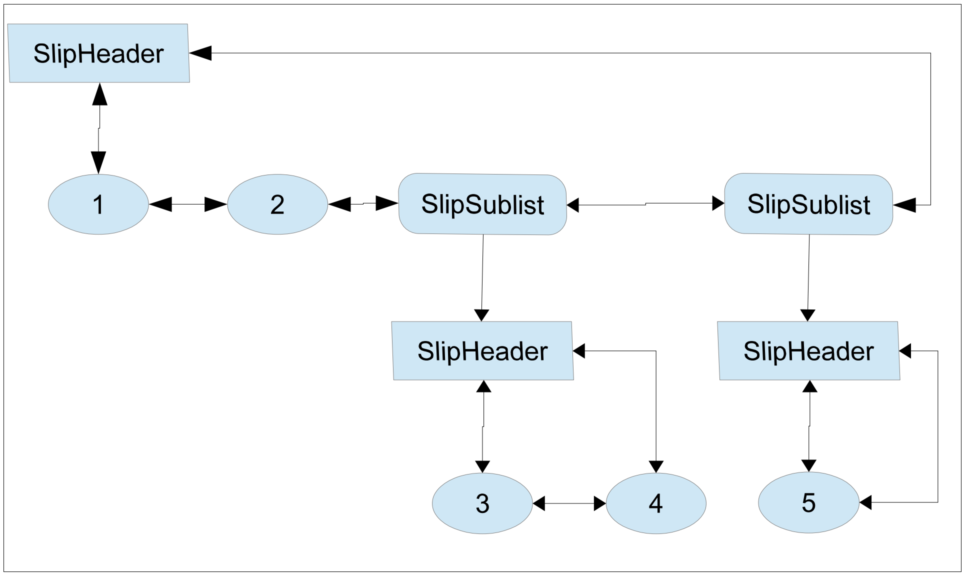

Figure 1.5-1 Functional List Representation

What is noted here is that each list ”**(1 2 (3 4) (5)), (3 4), and (5)**” is prefixed by a SlipHeader and that a reference to a list by another list is effected by a SlipSublist. All SLIP cells at the same list level have symmetric pointers and the SlipSublist reference to a list has a one-directional pointer. The effect of this is that from any one SLIP cell the application can go to the preceding or next cell but from a nested list the application can not return to the referencing SlipSublist cell. Once in a list you stay in a list, except when a SlipReader is used as a list iterator.

Figure 1.5-2 Reader Iterator is a functional representation of a SlipReader initialized to the outermost SlipHeader and advanced to the data cell containing '**3**'.

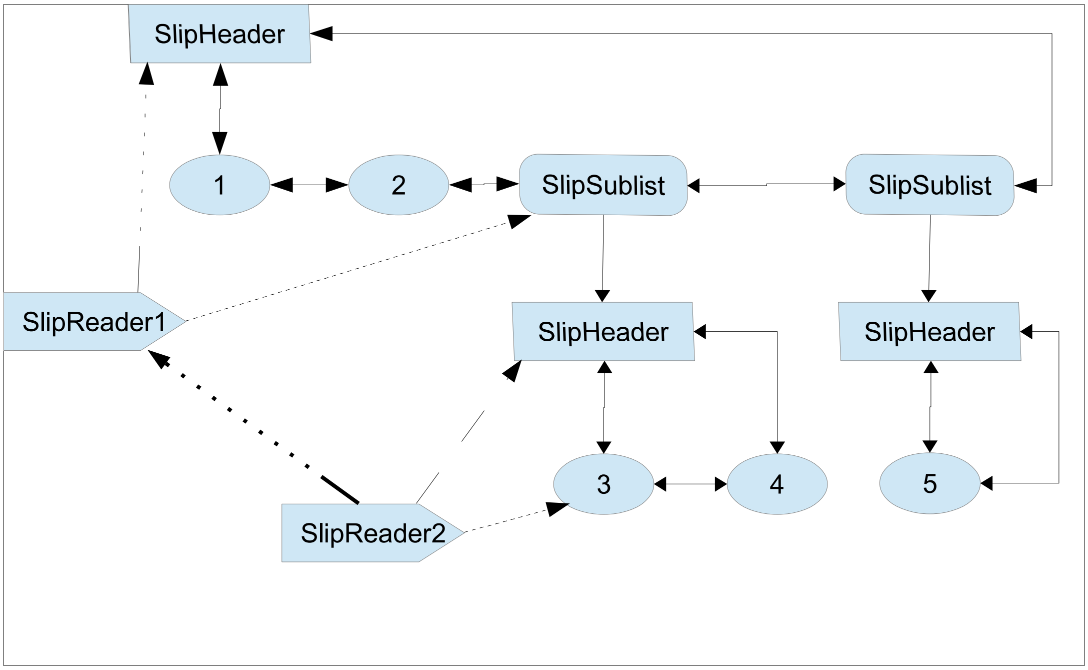

Figure 1.5-2 Reader Iterator

Figure 1.2-2 Reader Iterator represents:

1. SlipReader1: created when the SlipReader is created.
    a) The SlipHeader reference is to SlipHeader1 — · · ▶
    b) The SLIP cell reference is to SlipHeader1 ⋯⋯▶
2. Iterating through each cell until SlipSublist1 is reached causes the SLIP cell reference in the SlipReader to change. The SlipHeader reference remains on SlipHeader1.
3. On reaching SlipSublist1 a structural advance is made into the list represented by SlipHeader2, “**(3 4)**”, whereupon:
    a) SlipReader2 is created.
    b) The SlipHeader reference in SlipReader2 is initialized to SlipHeader2.
    c) The SLIP cell reference is initialized to SlipHeader2.
    d) The SlipReader backpointer references SlipReader1.
4. Advancing to “**3**” causes:

a) The SLIP reference pointer to reference the SLIP cell containing "**3**".

At this point there is sufficient information for SlipReader2 to return to the previous list by deleting SlipReader2 and restoring SlipReader1 as the current iterator. Not to fear, this is all done behind the curtain without application interaction.

We have an iterator that can enter lists and return from lists, and that avoids the overhead of a recursive entry.

Figure 1.5-3 Sequencer Iterator shows the same sequence of events when a SlipSequencer iterator is used.

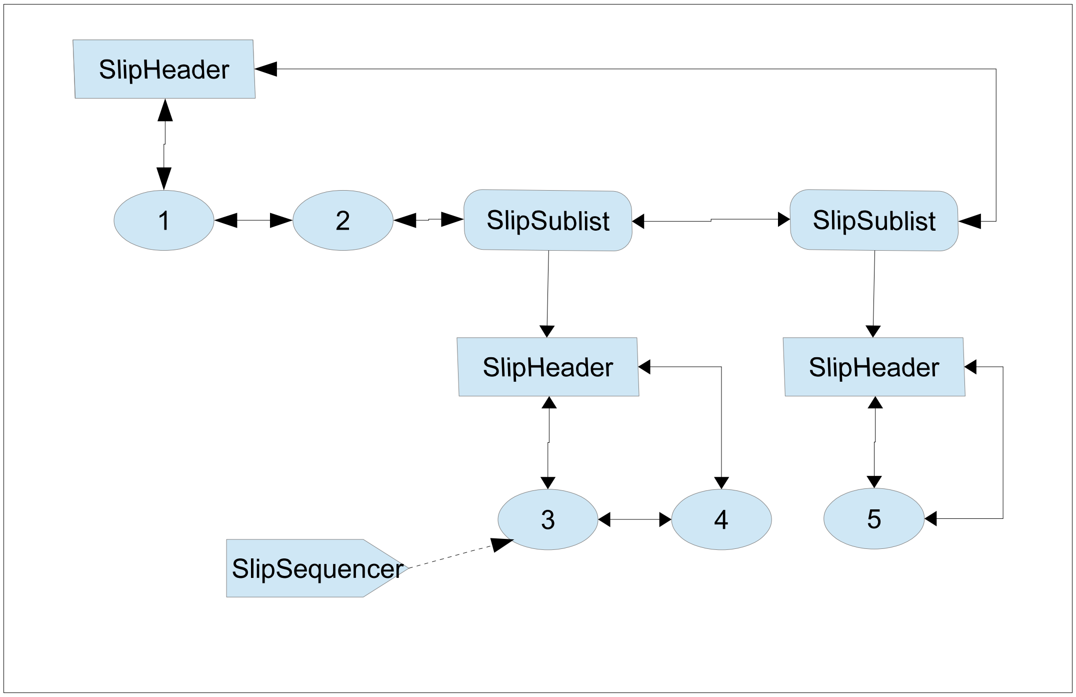

Figure 1.5-3 Sequencer Iterator

Iterating over the list used for Figure 1.2-3 we get.

1. SlipSequencer: created when the SlipSequencer is created.
    a) There is no SlipHeader reference.
    b) The SLIP cell reference is to SlipHeader1.

2. Iterating through each cell until SlipSublist1 is reached causes the SLIP cell reference in the SlipSequencer to change.
3. On reaching SlipSublist1 a structural advance is made into the list represented by SlipHeader2, "**(3 4)**", whereupon:
   a) The SLIP cell reference is modified to reference SlipHeader2.
4. Advancing to "**3**" causes:
   a) The SLIP cell reference to be modified ro reference the SLIP cell containing "**3**".

The SlipSequencer does not have a memory. Although somewhat faster in execution, there is no way for the iterator to know that it has entered a nested list or to return to the containing list.

In a word, the SlipReader has memory and the SlipSequencer does not.

1.6 Class Architecture

The class hierarchy forms the framework for discussion and revelation. The classes show the logical structure of objects and the relation between objects. The description of methods and the overarching reach of each method is given with respect to the class architecture.

Figure 1.6-1 SLIP Cell Architecture shows the class hierarchy of a list cell. The penultimate class, the SlipCellBase, is the base class for all objects extracted from the available space list (AVSL).

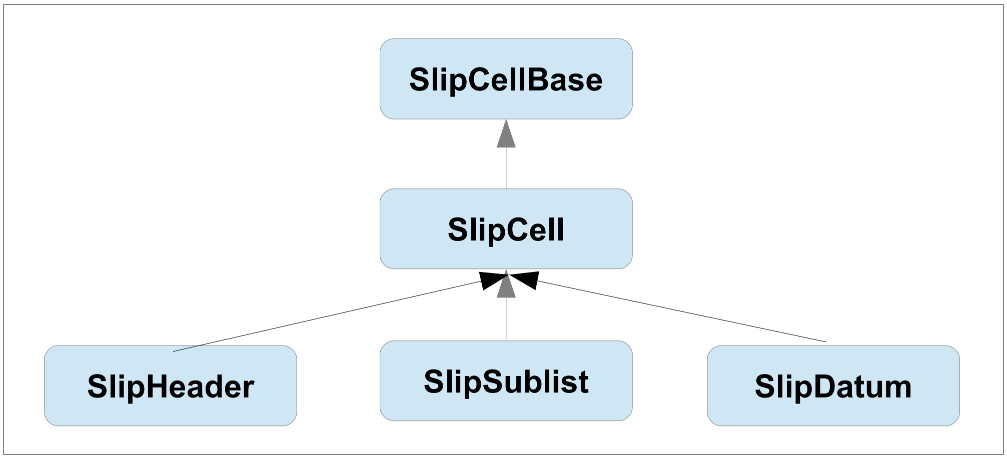

Figure 1.6-1 SLIP Cell Architecture

Methods contained in SlipCellBase deal with object properties and initialization and debug facilities for the AVSL. In SLIP there is no use of space other than the AVSL and hence, all SLIP objects use the SlipCellBase as a root class.

There are three SLIP list cells, the SlipHeader, the SlipSublist and the SlipDatum. The direct parent of these classes, the SlipCell, provides common functionality for list objects. Each list cell tailors the functionality, as required, for its own use and makes this tailored functionality available to the application.

Both the SlipCellBase and the SlipCell are pure abstract data types (ADT) and can not be instantiated. The Slip classes SlipHeader, SlipSublist, and SlipDatum are concrete classes available for instantiation.

- A SlipHeader object is a list header. Each list has one and only one SlipHeader and all functionality associated with list manipulation and handling as a list are contained in this class. The API includes some conveniences, some unique list functions, and maintenance and control over a unique associative list called the Descriptor List.
- A SlipSublist object contains a reference to a list. Although there is one and only one SlipHeader, hence all lists are unique, there may be many references to this list. Methods are provided to support maintenance and reasonable questions involving the referenced SlipHeader object.

  All instances of a SlipSublist object are unique. Where the same list (defined by its SlipHeader) can be on different lists or can be on the same list more than one time, each reference has a unique SlipSublist object.
- A SlipDatum object contains data. The data can be either C++ typed data (*bool*, *char*, *unsigned char*, *long*, *unsigned long*, and *double*) or application defined data (*string* and *PTR*). Methods to manipulate, cast, perform logical, arithmetic, bit, and unary operations are all contained here in a manner analogous to C++ operations on variables and literals. That is, “D + X is sufficient to specify addition involving a SlipDatum object and another SlipDatum object or a C++ object (literal or variable)..

Each SlipDatum object is unique. Where the same data type and value is needed more than once on the same list, or must appear on more than one list, each instance is a unique SlipDatum object.

All cells on a list must be from the AVSL. SlipHeader object and SlipSublist objects must be from the AVSL. But temporary SlipDatum objects may be on the runtime stack, either at application selection or created during expression evaluation by the SLIP system. No SLIP object can ever be created or used from the heap. None. Nada. Never. No.

Application defined data is a distinct data type with an associated data value contained in a SlipDatum object. A SlipDatum object is a carrier for all application defined data. Application defined data inherits from the class SlipPointer. There are two SLIP defined application data classes, SlipStringConst and SlipStringNonConst. The remaining class, Pointer, is for application use in defining application specific data to be inserted into lists.

The data in application defined data objects do not come from the AVSL. The application is required to acquire space for the data object, and to manage this space. SLIP will not manage application defined data space but defers all management to application provided methods. SLIP data deletion or creation of space for application defined data is done by using these call back methods.

It is important to be aware of data persistency in SLIP. All list cells are persistent outside of the context of any given application method. List cells exists as long as an application is executing. The application is required to delete lists, list objects, and list iterators when they are no longer needed. Application defined data is deleted by invoking call back methods. If application data is created within a method from the runtime stack, and the containing SlipDatum object is put on a ist the object data will be deleted during method exiting but the list object will not.. When the method is exited the SlipDatum object remains and references non-existent data. This is never very good and can be very bad. The application must ensure that data which is meant to be persistent is created from a persistent store.

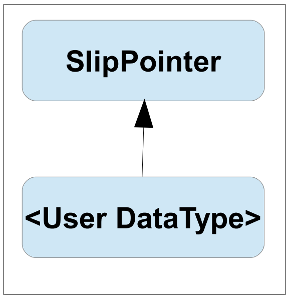

Figure 1.6-2 User Defined Data

Deleting a list containing non-persistent data will not necessarily delete the data at the time that the list deletion operation is requested. There is a latency built into SLIP which delays list cell deletion until the the AVSL is exhausted and more object space is needed. This delay is non-deterministic.

If you create application defined data, create it from the heap or from a persistent data store. Do not create application defined data on the runtime stack. The assumption by SLIP is that application defined data is persistent and that the application will delete the data explicitly.

The **SlipPointer** class is a pure abstract data type (ADT) and can not be instantiated. It contains the intrface methods required to inform the application when data deletion and copy are required, and support for returning a string representing raw application data and as a pretty-printed string. A User Data Type data handling and memory management is subject to application control.

2 .0 Data

2.1 Data Types

SLIP supports the data types given in Table 2.1-1 Data Types:

Table 2.1-1 Data Types

Data Type	Notes
bool	Boolean containing the value of **true** or **false**.
unsigned char	8-bit unsigned char (octet). 0 ≤ **range** ≤ 255
char	8-bit signed char (octet). −128 ≤ **range** ≤ 127
unsigned long	32-bit unsigned long. 0 ≤ **range** ≤ 4,294,967,295
long	32-bit signed long. −2,147,483,648 ≤ **range** ≤ 2,147,483,647
double	64-bit double. 2.2E-308 ≤ **range** ≤ 1.8E+308
string	<string> string.
PTR	Application defined pointer to a SlipPointer object.

The {*unsigned char*, *char*, *unsigned long*, and *long*} data types are called "**discrete data**". *Char* and *unsigned char* have a dual role as either a number or a UTF-8 character or both. *Double* is a real number and *string* and *PTR* are SLIP defined data types. *PTR* and string will be handled below.

A SlipDatum value with type *bool* has a value of '1' for **true** and '0' for **false**. Any operations which cause an assignment of a non-*bool* to a *bool* will cause the value the *bool* to have the value of '**0**' if the RHS is '**0**' or '**1**' for all other values. In any operation involving a *bool* on the RHS the value of the *bool* will be '**0**' for **false** and '**1**' for **true**. In any operation in which a bool is on the

LHS, it will be converted to '**1**' if the *bool* is **true** and '**0**' if the *bool* is **false** before the operation, and then converted to '**false**' if the result of the operation is '**0**' or '**true**' otherwise.

SLIP does not manage space on the heap or the runtime stack. SLIP only handles space on the AVSL. SLIP provides several means to support the application in handling *PTR* and string space.

Use of the other data types is subject to C++ conversion rules. If, for example, the application wants to assign the number '**1**' to a SlipCell, C++ will be amazingly obtuse if a type is not specifically included. That is in Example 2.1-1:

```
char X = 1;         // char variable with the value 1
SlipCell Y;         // SlipCell with no initial value
Y = X;              // type of X used
Y = 2;              // 2 can be any discrete type
Y = 1.2;            // type inferred as a double
```

Example 2.1-1 Data Types

C++ has no difficulty in inferring the type from the variable **X**. The type information is carried along with the variable during compilation and is used by C++ to determine what the type for **X** is, and hence, the required type for the assignment.

C++ can not infer the type of the literal '**2**'. The literal is ambiguous. The ambiguity arises because C++ doesn't know which one of the discrete data types (*unsigned char*, *char*, *unsigned long*, or *long*, and depending of the compiler, *bool*) is meant by '**2**'.. C++ will issue a nasty message until the application provides an explicit type cast The expression will work because

```
    Y = (long)2;
```

the C++ required type information is included.

The issue of ambiguity in recognizing literal discrete values is pervasive. The recommendation is to always use a type cast before a literal in any SLIP use. An explicit type cast avoids C++ compiler ambiguities.

2.2 Data Operations

If **Y** is a **SlipDatum** object and **X** is a **SlipDatum** object or a C++ variable or literal, then **Y** *op* **X** and **X** *op* **Y**, where *op* is an operator, are both legal expressions. **SlipDatum** objects are widened before the operation is performed; *bool*s and *char*s are converted to *long, unsigned char*s are converted to *unsigned long* and *doubles* are unconverted. The normal C++ operation rules apply.

*String*s, *PTR*s, **SlipHeader** and **SlipSublist** can only participate in the relational expressions == and *!=*. Comparisons involving a SlipHeader and a SlipSublist or two SlipSublists compare the SlipHeader referenced in the SlipSublist.

When a *bool* participates in an operation its value is '**1**' for **true** and '**0**' for **false**.

2.3 Simple Assignment

Assignment copies the right hand side (RHS) to the left hand side (LHS). The LHS cell remains valid after the copy in the sense that although the cell value and type changes the cell location does not. All pointers to the cell remain valid after the operation is complete.

Assignment is a like-to-like operation. In **`LHS = RHS`**, the SLIP cell type on the left hand side (LHS) must be the same as the SLIP cell type on the right hand side (RHS) with the exception that SlipSublist and SlipHeader cells are treated as referring to the SlipHeader.

`SlipSublist = SlipSublist` or **`SlipSublist = SlipHeader`** causes the SlipHeader reference count in the LHS to be decremented by one and the list deleted if the count is zero. The LHS SlipHeader reference is replaced by the RHS SlipHeader reference with the RHS SlipHeader reference count incremented by one. The referenced list is shared.

For **`SlipHeader = SlipHeader`** the LHS SlipHeader is flushed and then the RHS SlipHeader list is copied creating a new list. At the end of the operation the LHS SlipHeader list is a copy of the RHS SlipHeader list. The lists are not shared.

The list copy operation causes SlipDatum object to be copied and SlipSublist objects to be copied with the referenced SlipHeader reference count incremented by one. The copying operation is not recursive. Only the topmost list entries are copied.

For both the SlipHeader and SlipSublist if the list reference in the LHS is the same as the RHS, nothing is done.

SlipDatum object types are polymophic during assignment. An assignment operation between two SlipDatum cell types, `X = Y`, is equivalent to `(cast)X = Y.` Where the cast casts the object type of `X` into the object type of `Y`. In C++ the LHS data type is invariant. In SLIP it is not.

If the LHS SlipDatum object is a *string* or *PTR* then the data is deleted before the assignment. If the RHS is a *string* or *PTR* then the value is copied. The exact meaning of deletion and copy are explained when application data types are discussed.

Summarizing this in Table 2.3-1 Assignment Operations:

Table 2.3-1 Assignment Operations

LHS			**=**	**RHS**			**Format**	**Description**
H	**S**	**D**		**H**	**S**	**D**	**LHS = (type)RHS**	
X				X			H1 = H2;	Flush H1 and copy H2 to H1
X					X		H1 = S;	Flush H1 and copy S.H2 to H1
	X			X			S = H2;	Flush S.H1 and copy H2 to S.H1
	X				X		S1 = S2;	Flush S1.H1 and copy S2.H2 to S1.H1
		X				X	D1 = D2;	Perform actions as if D2 were a literal
		X					D = (*bool*)X;	Delete the old SlipDatum value and replace it with X.
		X					D = (*unsigned char*)X;	Delete the old SlipDatum value and replace it with X
		X					D = (*char*)X;	Delete the old SlipDatum value and replace it with X
		X					D = (*unsigned long*)X;	Delete the old SlipDatum value and replace it with X
		X					D = (long*)*X;	Delete the old SlipDatum value and replace it with X
		X					D = (*double*)X;	Delete the old SlipDatum value and replace it with X
		X					D = (*string&*)X;	Delete the old SlipDatum value and copy X
		X					D = (*PTR*)X;	Delete the old SlipDatum value and copy X

Legend

- H: SlipHeader cell
- S: SlipSublist cell
- D: SlipDatum cell
- X: literal data value

Example 2.3-1 Assignments provides some living, working examples of assignments.

```
SlipHeader L1;   // magically (1 2 (3 4) 5)
SlipHeader L2;   // magically (5 (4 3) 2 1)
SlipHeader L3;   // magically (5 6 7 8 9 0)
SlipReader R1(L1); // points to L1
SlipReader(R3(L3); // points to L3
L1 = L3;         // L1: (5 6 7 8 9 0)
R1.advanceLWR(); // R1 → 5
R1 = L3;         // ERROR: '5' can not be replaced by a list
R1 = (double)1.3;// L1: (1.3 6 7 8 9 0)
R1 = “str”;      // L1: (“str” 6 7 8 9 0)
R2.advanceLNR(); // R1 → (4 3)
R2 = L3;         // L2: (4 (5 6 7 8 9 0) 2 1)
L1.deleteList();
L2.deleteList();
l3.deleteList();
```

Example 2.3-1 Assignments

2.4 Relational Operators

The relational operators always succeed. If the test being performed is illegal or if the result of the test is not **true**, then the value of a comparison is **false**.

The relational operators can be divided into equal (==) and unequal (!=) and everything else. Equal and unequal work for all data types without exception. The remainder of operators work for the SlipDatum objects and their operation depends on the object data types.

SlipHeader cells and pointers to SlipHeader cells are equal if they represent the same list. SlipSublists are equal if they refer to the same SlipHeader cell, and a SlipSublist and SlipHeader cell are equal if the SlipSublist cell refers to the SlipHeader cell. In all other cases the the result of the comparison is **false**.

In a SlipDatum object, for all data types except *PTR*, the implementation of all comparisons is identical to the C++ standard. Casts and return values are the same as defined in the C++ standard for all data types.

For *PTR* types equal and unequal are supported as a default, and only between *PTR* data types. If two SlipDatum objects with *PTR* data types are compared, they are equal if the application data object referenced in the SlipDatum objects are the same, otherwise they are unequal. As a default, a *PTR* data type is uncastable. See the *PTR* section for more details on expanding the comparisons.

String comparisons are based on the default C++ standard. In all cases, if the C++ standard accepts a relational comparison between *string*s, then SLIP accepts the same comparison and yields the same results. Note that the collating sequence of values is locale sensitive. The standard locale used in the comparison is UTF-8, English.

If two SlipDatum objects are compared, or if a SlipDatum object and a literal are compared, then **true** will be returned if the LHS can be cast to the same data type as the RHS, sic., the RHS cans be cast to the same data type as the LHS, and the comparison is successful. In this case we say that the the data type and the relation are “satisfiable”.

A SlipDatum object and a literal can be compared in any order. If '**op**' is a relational operator and **D** is a SlipDatum object, then both **D op X** and **X** op **D** are valid. Table 2.4-1 Relational Operators summarizes these results. Where **D op X** is shown, **X** op **D** can be substituted.

Table 2.4-1 Relational Operators

LHS			**op**	**RHS**			**Format**	**Description**
H	**S**	**D**		**H**	**S**	**D**	**LHS = (type)RHS**	
X			==	X			H1 == H2;	**true** if H1 and H2 are the same SlipHeader cell
X			==		X		H1 == S;	**true** if H1 and S.H2 are the same SlipHeader cell
	X		==	X			S == H2;	**true** if S.H1 and H2 are the same SlipHeader cell
	X		==		X		S1 == S2;	**true** if S1.H and S2.H reference the same list
X			==			X	H1 == D2;	**false** illegal comparison
	X		==			X	S1 == D2;	**false** illegal comparison
		X	==	X			D1 == H1;	**false** illegal comparison

Table 2.4-1 Relational Operators

LHS			**op**	**RHS**			**Format**	**Description**
H	**S**	**D**		**H**	**S**	**D**	**LHS = (type)RHS**	
		X	==		X		D1 == S1;	**false** illegal comparison
		X	op			X	D1 op D2;	**true** for data type and relational satisfiability
		X	op				D op (*bool*)X;	**true** for data type and relational satisfiability
		X	op				D op (*unsigned char*)X;	**true** for data type and relational satisfiability
		X	op				D op (*char*)X;	**true** for data type and relational satisfiability
		X	op				D op (*unsigned long*)X;	**true** for data type and relational satisfiability
		X	op				D op (long)X;	**true** for data types and relational satisfiability
		X	op				D op (*double*)X;	**true** for data types and relational satisfiability
		X	op				D op (*string*)X;	**true** for data types and relational satisfiability
		X	op				D op (*PTR*)X;	**true** for data types and relational satisfiability

Legend

- H: SlipHeader cell.
- S: SlipSublist cell.
- D: SlipDatum cell.
- X: literal data value.
- == any one of '==' and '!='
- op any one of {<, <=, ==, >= ,>, !=}

2.5 Casting

Casting is available for all data types except *string* and *PTR*. Casting of a *bool*, *char*, *unsigned char*, *long*, *unsigned long*, and *double* works the same as in C++. The operation can be from a SlipDatum object to another, or from a SlipDatum object to a application defined variable. In Example 2.5-1 Casting, the value after casting is 15 in the data type cast for all discrete types.

```
SlipDatum D((long)15);
bool          b  = (bool)D;          // true
char          c  = (char)D;          // 15
unsigned char uc = (unsigned char)D; // 15
long          l  = (long)D;          // 15
unsigned long ul = (unsigned long)D; // 15
double        d  = (double)D;        // 15.0
```

Example 2.5-1 Casting

The *bool* value **true** represents the value of '1' after casting and it has an stored representation of '1' in the variable 'b'. Casting to a *double* yields a double value of (approximately) 15.0.

2.6 Unary Operations

The allowed unary operations are {+, - , ++ prefix, ++ suffix, -- prefix, -- suffix, ~, and ! }.

The SlipDatum data types of *string* and *PTR* default can participate in a unary operation. This default behavior can be changed for a *PTR* but not for a *string*. SlipHeader and SlipSublist cells do not have unary operations, *double* can only participate in unary + and -, and bool can only participate in unary +, -, !, and ~. This is summarized in Table 2.6-1 Unary Operators below

Table 2.6-1 Unary Operators

		discrete data types							
op	**bool**	**char**	**uChar**	**long**	**uLong**	**Double**	**String**	**PTR**	**result**
~	X	X	X	X	X				(*ulong*) bit not. (bool) 0xFFFF FFFF or 0x0
!	X	X	X	X	X				**(***bool***) true** if > 0 or (*bool*)**false**
+	X	X	X	X	X	X			(*SlipDatum*(T)) unchanged data value
-	X	X	X	X	X	X			(*SlipDatum*(T)) arithmetic negation
++p		X	X	X	X				(*SlipDatum*(T)) ++D
--p		X	X	X	X				(*SlipDatum*(T)) --D
++s		X	X	X	X				(*SlipDatum*(T)) D++
--s		X	X	X	X				(*SlipDatum*(T)) D--

Legend

- ~: bit not. *Bools* return 0xFFFF FFFF if **true** or 0x0 if **false**
- **op:** unary operation
- T: SlipDatum data type – any of the allowed data types. Unchanged after operation completion
- D: SlipDatum((T))D data cell with type T. 'T' is unchanged after operation completion

2.7 Binary Operations

The SlipDatum data types of *string* and *PTR* default is to not participate in a binary operation. This default behavior can be changed for a *PTR* but not for a *string*. Data type *double* cannot participate in modulus (%) operations. SlipHeader and SlipSublist cells do not have binary operations.

SlipDatum objects can participate in a binary operation as the LHS, the RHS or both LHS and RHS of an operation. The operation involves the data type and data value stored in the SlipDatum object(s) and will yield the same result as C++ with the same data types and values. The operations supported are {+, -, *, /, and %}. SlipDatum data type double does not have a modulus (%) operation.

If either the LHS or RHS of an expression is a *double* data type then the resultant data type is double. In all other cases the data type of the result is the same as C++ would return in operating on the data types.

This is summarized in Table 2.7-1 Binary Operators.

Table 2.7-1 Binary Operators

		discrete data types							
op	**bool**	**char**	**uChar**	**long**	**uLong**	**Double**	**String**	**PTR**	**result**
+	X	X	X	X	X	X			*SlipDatum*((T))
-	X	X	X	X	X	X			*SlipDatum*((T))
*	X	X	X	X	X	X			*SlipDatum*((T))
/	X	X	X	X	X	X			*SlipDatum*((T))
%		X	X	X	X				(*long*)

Legend

- **op:** unary operation
- T: SlipDatum data type – any of the allowed data types. The resultant data type depends on the underlying data types involved in the operation

```
SlipDatum X((ulong)20);
SlipDatum Y((char)7);
long          a = 15;
unsigned char b = 17;

X + Y;          // 27
X % a;          // 6
b % y;          // 3
```

Example 2.7-1 Binary Operation

Example 2.7-1 provides a digestible view of some binary operations. The resultant data type for the modulus operations is (*long*). The additional data type depends on the C++ standard.

2.8 Bit and Shift Operations

Bit operations are restricted to discrete types and *bool*s. *Doubles*, *string*s, and *PTR*s are invalid. The LHS, RHS, or both must be a SlipDatum object of the appropriate type. The resultant data type of these operations is a *long* if the LHS is a SlipDatum object. Literals are cast to a *long* before use, and *bool*s have a value of '**0**' for **false** and '**1**' for **true**.

Shifting operations treat the input as unsigned, the sign bit is ignored. A right shift shifts the sign bit and replaces bits shifted right by '**0**', and a left shift shifts through the sign bit and replaces bits shifted left by '**0**'.

The operations and their result is given in table Table 2.8-1 Bit and Shift Operators.

Table 2.8-1 Bit and Shift Operators

op	Description	Example
Y << X	Shift Y left by X bits	0xFFFF FFFF << 3 → 0xFFFF FFF8
Y >> X	Shift Y right by X bits	0xFFFF FFFF >> 3 → 0x1FFF FFFF
Y & X	Bit and Y and X	0x2222 1010 & 0x1234 5678 → 0x0220 1010
Y | X	Bit or Y and X	0x2222 1010 | 0x1234 5678 → 0x3236 5678
Y ^ X	Bit exclusive or Y and X	0x2222 1010 ^ 0x1234 5678 → 0x3026 4668

Legend

- Y: either a SlipDatum discrete or *bool* object or a literal
- X either a SlipDatum discrete or *bool* object or a literal

2.9 Complex Assignments

A complex assignment consists of a binary, bit or shift operation symbol followed by an equal sign. The legal combination of symbols are give in Table 2.9-1 Complex Assignments.

Table 2.9-1 Complex Assignments

op	description	allowed data types	example
+=	Add and assign	*bool*, discrete data types, *double*	D += X
-=	Subtract and assign	*bool*, discrete data types, *double*	D -= X
*=	Multiply and assign	*bool*, discrete data types, *double*	D *= X
/=	Divide and assign	*bool*, discrete data types, *double*	D /= X
%=	Take modulus and assign	*bool*, discrete data types	D %= X
<<=	Shift left and assign	*bool*, discrete data types	D << X
>>=	Shift right and assign	*bool*, discrete data types	D >>= X
&=	Bit and and assign	*bool*, discrete data types	D &= X
|=	Bit or and assign	*bool*, discrete data types	D |= X
^=	Exclusive or and assign	*bool*, discrete data types	D &= X

Legend

- D: SlipDatum object

- X SlipDatum object, literal, or expression

The LHS can be a C++ variable or a SlipDatum object. The LHS SlipDatum data type is invariant. The operation will cause a change of the SlipDatum value but not the SlipDatum data type. The SlipDatum value takes part in the operation, which can be decomposed into D = D **op** X, with D and X with the same meaning as in Table 2.9-1 Complex Assignments and **op** being one of the operations in the table stripped of the concatenated equal sign (=).

SlipHeader, SlipSublist and *string* objects are never legal. *Double* is not legal for bit and shift operations. SlipDatum data type *PTR* objects default iof illegal can be overriden.

If the LHS side is a variable and the result of expression evaluation of the RHS is a SlipDatum object, then the SlipDatum object is cast to the data type of the LHS before **op** evaluation begins. That is: V **op** D becomes V **op** (cast)D, where **op** is as above.

3 .0 Common Operations

There are certain operations that are shared in common amongst the SLIP cells (SlipHeader, SlipSublist, SlipDatum) directly and indirectly with the SLIP iterators (SlipReader, SlipSequencer). These operations include interrogatories (are you?), assignment, replacement, and logical operations.

Where applicable, operations available to the iterators are operations on the SLIP cell referenced by the iterator and not the SlipHeader. They are not operations on the iterator itself.

3.1 Interrogatories

All questions return a boolean (**true** or **false**). A **true** implies that the question is answered successfully, and a **false** if the cell does not satisfy the question asked. Every non-iterator SLIP cell has the interrogatories as a cell property.

The interrogatories apply to SlipHeaders, SlipSublists, and SlipDatum objects. When iterators are used, then the SLIP cell referenced by the iterator is interrogated. The properties are owned by the cell type not the cell. They are static in the sense that any object of a given cell type will give the same answer to the same question.

The interrogatories are appropriately called the object properties. The format for method invocation is ((SlipCellBase&)**X).method()**. The are listed in Table 3.1-1 Interrogatories.

Table 3.1-1 Interrogatories

return	((SlipCellBase&)X).method()	Description
bool	isAVSL(const SlipCellBase*)	Is the argument from the AVSL
bool	isData()	Is this a SlipDatum cell.
bool	isDeleted()	Has the cell been deleted (restored to the AVSL).
bool	isDiscrete()	Is this a SlipDatum cell with a discrete number.
bool	isHeader()	Is the cell a SlipHeader.
bool	isName()	Is the cell a SlipSublist.
bool	isNumber()	Is this a SlipDatum cell with a number.
bool	isPtr()	Is the cell a application defined SlipDatum cell.
bool	isReal()	Is this a SlipDatum cell with a real.
bool	isString()	Is this a SlipDatum cell for a string.
bool	isSublist()	Is this a SlipSublist cell.
bool	isTemp()	Is this cell on the runtime stack.
bool	isUnlinked()	Is this cell not part of a list.

Note that **isAVSL()** requires a cell pointer to be input as an argument. Its invocation format is **isAVSL(X)**.

In common usage the application creates a SLIP cell object (cell) and asks about cell properties. For example:

if (((SlipCellBase&)cell.)isReal()) … or
if (cell.isReal()) …

In this case the answer is '**true**' if the cell is a SlipDatum cell and if the datum type is real (type double). If the cell is not a SlipDatum cell or if the datum type is not real then the answer is **false**.

Some special considerations are required for “isAVSL()”. The question being asked is whether the input cell is from the AVSL or the runtime stack or the heap. If it is from the AVSL then the response is **true**. In all other cases the response is false. The cell itself may or may not be in a list.

3.2 Replacement

We will call the object being replaced as the LHS and the source of the replacement, the RHS. In a manner analogous to assignment, LHS = RHS implies that the LHS object data type and data value will be replaced by the RHS object data type and data value. In assignment, the LHS data value is replaced by the RHS data value.

The LHS must be either a SlipSublist or SlipDatum object. The RHS can be a SlipSublist, SlipDatum, SlipHeader object or literal value.

The LHS must be on a list. The RHS may or may not be on a list.

Replace is a superset of simple assignment. It replaces like-to-like and unlike-to-unlike SLIP cell types. When like-like replacements are done the result is the same as assignment. In unlike-to-unlike both the data type and the data value change. In this mode there is no guarantee that the SLIP object will not be deleted and references to the object are not reliable.

The valid LHS and RHS relations are:

- If a literal value is used, a new SlipDatum object is created and inserted into the list in place of the LHS. The LHS object is deleted.
- If the LHS and RHS are type incompatible (one a SlipSublist and the other a SlipDatum) then a new Slip object is created and inserted into the list and the LHS object deleted.
- If the LHS is a SlipSublist and the the RHS is either a SlipSublist or SlipHeader, then the LHS SlipHeader reference count is decremented and the RHS SlipHeader reference inserted and reference count incremented.

If the replacement creates a new object, the old object will be deleted and the new object inserted into the list at the same location.

The SLIP API is given in Table 3.2-1 Replace Methods. All methods return a reference (SlipCell&) to the new LHS list object. Method invocation is ((SlipCell&)**X).method()**.

Table 3.2-1 Replace Methods

return	**((SlipCell&)X).method()**.	**Description**
SlipSublist&	replace(const SlipHeader&)	Creates a new SlipSublist object
SlipSublist&	replace(const SlipSublist&)	Creates a new SlipSublist object
SlipDatum&	replace(const SlipDatum&)	Uses input as a model
SlipDatum&	replace(*bool)*	Creates SlipDatum *bool* object
SlipDatum&	replace(*char*)	Creates SlipDatum *char* object
SlipDatum&	replace(*unsigned char*)	Creates SlipDatum *uChar* object
SlipDatum&	replace(*long*)	Creates SlipDatum *long* object
SlipDatum&	replace(*unsigned long*)	Creates SlipDatum *uLong* object
SlipDatum&	replace(*double*)	Creates SlipDatum *double* object
SlipDatum&	replace(const *PTR*, const void* operation=**null**)	Creates SlipDatum *PTR* object
SlipDatum&	replace(const *string*&, bool constFlag=**false**)	Creates SlipDatum *string* object
SlipDatum&	replace(const *string**, bool constFlag=**false**)	Creates SlipDatum *string* object

The *PTR* and *string* 2nd arguments are defaulted. They will be explained in Section 7, User Data Types.

Some representative code is given in Example 3.2-1: Replacement.:

```
                        //magically initialized to (1 2 (3 4) 5)
SlipHeader L1 = new SlipHeader();
                        //magically initialized to (5 (4 3) 2 1)
SlipHeader L2 = new SlipHeader();)
SlipReader R1(L1);
SlipReader R2(L2);
R1.advanceLNR();
R2.advanceLWR().advanceLWR();
R1.replace(R2);          // L1: (1 2 (4 3) 5)  (3 4) deleted
R1.replace((bool)false));// L1: (1 2 false 5)  (4 3) deleted
R1.replace((double)5.3));// L1: (1 2 5.3 5)    false replaced
R1.replace(“str”);       // L1: (1 2 “str” 5)  4.3   replaced
L1.deleteList();
l2.deleteList();
```

Example 3.2-1: Replacement

3.3 Insert Operations

SLIP objects and literals can be inserted to the left or right of the current cell. We designate the current cell as the LHS of the operation and the object to be inserted as the RHS. The following conditions apply:

- The LHS must either be a SlipHeader object or on a list.
- The RHS may not be on a list but must be from the AVSL and not the runtime stack,.
- The RHS may be a SlipHeader, SlipSublist, or SlipDatum object or a literal.
  - Literals cause a new SlipDatum object to be created.
  - A SlipHeader object causes a SlipSublist object to be created referencing the SlipHeader and the SlipHeader reference count is incremented. The new SlipSublist object is inserted into the list.
  - A SlipSublist object is inserted into the current list. The referenced SlipHeader reference count is not changed.
  - A SlipDatum object is inserted into the current list.
- No SlipSublist or SlipDatum object can be on more than one list. A SlipSublist or SlipDatum object need not be on any list.

Valid list cells are SlipHeader, SlipSublist, and SlipDatum object. Each list has one and only one SlipHeader. Each list can have zero of more SlipSublist and SlipDatum objects, however, each SlipSublist and SlipDatum object is unique. A list cell can not co-exist on two or more lists, and although two or more SlipSublist objects can reference to the same SlipHeader (list), the objects are unique.

An item can be inserted to the left or right of an existing item. The API is given in Table 3.3-1 Insert Methods. All methods return a reference to the inserted object. Method invocation is ((SlipCell&)X).method**()**.

Table 3.3-1 Insert Methods

return	**((SlipCell&)X).method()**	**Description**
SlipSublist&	insLeft(SlipHeader&)	Create & insert a SlipSublist object
SlipSublist&	insLeft(SlipSublist&)	Copy & insert a SlipSublist object
SlipDatum&	insLeft(SlipDatum&)	Copy & insert a SlipDatum object
SlipDatum&	insLeft(*bool*)	Insert a **new** SlipDatum *bool* cell
SlipDatum&	insLeft(*char*)	Insert a **new** SlipDatum *char* cell
SlipDatum&	insLeft(*unsigned char*)	Insert a **new** SlipDatum *uchar* cell
SlipDatum&	insLeft(*long*)	Insert a **new** SlipDatum *long* cell
SlipDatum&	insLeft(*unsigned long*)	Insert a **new** SlipDatum *ulong* cell
SlipDatum&	insLeft(*double*)	Insert a **new** SlipDatum *double* cell
SlipDatum&	insLeft(const *PTR*, const void* operation=**default**)	Insert a **new** SlipDatum *PTR* cell
SlipDatum&	insLeft(const string&, bool constFlag=**false**)	Insert a **new** SlipDatum *string* cell
SlipDatum&	insLeft(const string*, bool constFlag=**false**)	Insert a **new** SlipDatum *string* cell
SlipSublist&	insRight(SlipHeader&)	Create & insert a SlipSublist object
SlipSublist&	insRight(SlipSublist&)	Copy & insert a SlipSublist object
SlipCell&	insRight(SlipDatum&)	Copy & insert a SlipDatum object
SlipDatum&	insRight(*bool*)	Insert a **new** SlipDatum *bool* cell
SlipDatum&	insRight(*char*)	Insert a **new** SlipDatum *char* cell
SlipDatum&	insRight(*unsigned char*)	Insert a **new** SlipDatum *uchar* cell
SlipDatum&	insRight(*long*)	Insert a **new** SlipDatum *long* cell
SlipDatum&	insRight(*unsigned long*)	Insert a **new** SlipDatum *ulong* cell
SlipDatum&	insRight(*double*)	Insert a **new** SlipDatum *double* cell
SlipDatum&	insRight(const PTR, const void* operation=**default**)	Insert a **new** SlipDatum *PTR* cell
SlipDatum&	insRight(const string&, bool constFlag=**false**)	Insert a **new** SlipDatum *string* cell
SlipDatum&	insRight(const string*, bool constFlag=**false**)	Insert a **new** SlipDatum *string* cell

```
SlipHeader L1 = new SlipHeader();           // create (1 2 (3 4) 5)
SlipHeader L2 = new SlipHeader();           // create (5 (4 3) 2 1)
SlipHeader S1 = new SlipHeader();           // create (3 4)
SlipHeader S2 = new SlipHeader();           // create (4 3)
long one           = 1;
unsigned long two  = 2;
char three         = 3;
unsigned char four = 4;
S1.insLeft((char)3);                        // (3)
S1.insLeft(four);                           // (3 4)
S2.insRight(three);                         // (3)
S2.insRight((long)4);                       // (4 3)
L1.insLeft(one);                            // (1)
L1.insLeft(two);                            // (1 2)
L1.insLeft(S1);                             // (1 2 (3 4))
L1.insLeft((long)5);                        // (1 2 (3 4) 5)
L2.insRight((long)1);                       // (1)
L2.insRight((long)2);                       // (2 1)
L2.insRight(S2);                            // ((4 3) 2 1)
L2.insRight((long)5);                       // (5 (4 3) 2 1)
L1.deleteList();                            // list restored to AVSL
L2.deleteList();                            // list restored to AVSL
S1.deleteList();                            // list restored to AVSL
S2.deleteList();                            // list restored to AVSL
```

Example 3.3-1 Insert Operations

Some examples of use are given in Example 3.3-1 Insert Operations.

The variables (one, two, three, four) are typed by C++. This data type is used directly in the method to determine the data type to use in creating a SlipDatum((T)) object. The data type T is the data type of the variable, and the data value is the data value. When an integer literal is used directly in the insert method the data type is ambiguous (it can be *char*, *unsigned char*, *long*, or *unsigned long*). The application must provide a cast. Sorry, this is C++.

The **insLeft()** methods when used with a SlipHeader object is equivalent to an enqueue operation onto a list. The **insRight()** methods when used with a SlipHeader object is equivalent to a push operation onto a list. When used with a list cell, it is just an insert.

We can create S1 above by doing:

```
S1.insLeft(three).insRight(four);
```

chaining the insert commands together. Since S1 is initially an empty list, the application of insLeft() puts 'three' on the top of the list returning a reference to 'three' and insRight() puts 'four' to the right of 'three' creating a list, S1:(3 4).

This will work the same way when a list is not empty. In this case **insLeft()** will place a cell on the bottom of the list and **insRight()** to its right.

In the context of an insert operation, a SlipHeader behaves the same as a list cell. When performing an insert on a list cell, **insLeft()** puts the new cell before the current cell and **insRight()** puts the new cell after the current cell. And so for a SlipHeader the normal names for these operations is enqueue and push respectively.

3.4 Move Operations

Move a cell or a group of cells from one list to another. The destination cell must be on a list, the source cell may be on a list. The move operations change the location of list cells, they do not change the list cell values. Hence, they are not equivalent to either an assignment or a replace.

To move an individual cell from one list to another, **moveLeft()** and **moveRight()** are the methods to use. If the source cell in on a list, it will be unlinked and moved the the left or right of an existing cell in the destination list. If the source cell is not on a list then the move will act as an insert.

To move an entire list from one list to another, **moveListLeft()** and **moveListRight()** are the methods to use. All cells in the source list are moved to the left or right of a cell in the destination list. At the end of the operation, the source list will be empty.

The source list for a **moveListLeft()** and **moveListRight()** can be either a list header, SlipHeader object, or a reference to a list, a SlipSublist object. In either case the operations are the same.

This is summarized in Table 3.4-1 Move Methods.

Table 3.4-1 Move Methods

return	((SlipCell&)X).method()	Description
SlipCell&	moveLeft(SlipCell&)	Move the source object to the left of the destination cell
SlipCell&	moveRight(SlipCell&)	Move the source object to the right of the destination cell
SlipCell&	moveListLeft(SlipHeader&)	Move the source list to the left of the destination cell
SlipCell&	moveListLeft(SlipSublist&)	Move the referenced list to the left of the destination cell
SlipCell&	moveListRight(SlipHeader&)	Move the source list to the right of the destination cell
SlipCell&	moveListRight(SlipSublist&)	Move the referenced list to the right of the destination cell

Example 3.4-1 Move Operations shows the move operations.

```
                            //magically initialized to (1 2 (3 4) 5)
SlipHeader   L1 = new SlipHeader();
                            //magically initialized to (5 (4 3) 2 1)
SlipHeader   L2 = new SlipHeader();
SlipDatum&   c1 = *(L1.getRightLink();)  //  '1' in L1
SlipSublist& c2 = *(L2.getRightLink()->getRightLink()); // (4 3) in L2
                                     // L1: (2 (3 4) 5)
c2.moveLeft(c1);                     // L2: (5 1 (4 3) 2 1)
                                     // L1: (2 (3 4) 5)
c1.moveLeft(c2);                     // L2: (5 (4 3) 1 2 1)
                                     // L1: (5 (4 3) 1 2 1 2 (3 4) 5)
L1.moveListRight(L2);                // L2: ( )
L1.deleteList();
L2.deleteList();
```

Example 3.4-1 Move Operations

Explaining the self-explanatory example, we assume L1 and L2 have been initialized without going into the mechanisms of how.

- The SlipDatum variables, 'c1' and 'c2', are initialized using the method **getRightLink()**. This method retrieves the 'next cell pointer' for a SlipCell and returns its value, a pointer to the next cell. The pointer is dereferenced and assigned.

The interesting thing is that the 'cell' being referenced by 'c2' has granularity. The cell is a SlipSublist type referencing a list. But, the list referenced is not 'seen' in this example, only the container of the reference. It is an artifice of construction to show the contents of the referenced list. There are various ways to enter or access the referenced list, but this example does not require that knowledge.

The good news is that **getRightLink()** is there for use, the bad news is that it gives unlimited access to SLIP internals and should be used with caution.

- The single cell referenced by 'c2' is moved to the left (preceding) the single cell referenced by 'c1'. The cells were respectively located in list L2 and L1 and are now located in L2.
- The single cell referenced by 'c1' is moved to the left (preceding) the single cell referenced by 'c2'. And now we observe a strange thing. Whereas before we moved a cell between two lists, now we move a cell in the same list. Perfectly legal. The requirement is that the cells be in a list, not that they be in different lists. This move changes the position of 'c1' from preceding 'c2' to following 'c2' This could have been done in a single step by using **c2.moveRight(c1)**, but then there goes the example.
- The entire list L2 is moved to the right (following) the header of list L1. In other words, list L2 is pushed onto L1. At the end of the operation L2 is empty and L1 contains both lists.

3.5 Debug Tools

Sparse and skinny. There are some tools available to output characteristics of the SLIP system. At the moment, there are no analytical tools. The analysis is left to the application. In time this should change.

Given that the Slip.h header file is included, Table 3.5-1 Debug Methods lists the available methods.

Table 3.5-1 Debug Methods

<table>
<tr><th>return</th><th>function()</th><th>Description</th></tr>
<tr><td><i>void</i></td><td>avslHistory(bool)</td><td><b>true</b> start, <b>false</b> stop, initially false. Logs each insertion and deletion event in the AVSL.</td></tr>
<tr><td>SlipState</td><td>getSlipState()</td><td>Returns internal state of SLIP.
<table>
<tr><th>Field Names</th><th>Description</th></tr>
<tr><td>avail</td><td>Total AVSL free cells (not accurate)</td></tr>
</table></td></tr>
</table>

Table 3.5-1 Debug Methods

<table>
<tr><th>return</th><th>function()</th><th>Description</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><td>alloc</td><td>Initial allocation (in cells)</td></tr>
<tr><td>delta</td><td>Incremental allocation (in cells)</td></tr>
<tr><td>total</td><td>Total AVSL cells – both free and used</td></tr>
</table>
</td></tr>
<tr><td><i>void</i></td><td>printAVSL(caption)</td><td>Pretty-print the AVSL state and AVSL list.
The AVSL state variables are output followed by a dump of each cell in the AVSL list. The AVSL list cell dump includes an ordinal number representing the cell position in the AVSL list, with the first position being output as one ('1').
Output of the AVSL list is in order of appearance within the list. Each cell is analyzed before output and diagnostics messages are issued as required. Identifying which fragment contains the cell is not done.
<b>WARNINGS</b>
<ul>
<li>**** Left link is not 0xdeadbeef</li>
<li>**** Sublist header pointer is NULL</li>
<li>**** Sublist header pointer does not point to a header</li>
<li>**** Header reference count > 0. Header is still active</li>
<li>**** Number of cells in AVSL does not agree with free space count</li>
</ul>
</td></tr>
<tr><td><i>void</i></td><td>printClassSizes()</td><td>Outputs the size in bytes of each SLIP class.</td></tr>
<tr><td><i>void</i></td><td>printMemory(caption)</td><td>Output all cells in each AVSL fragment. A combination of <b>printFragmentList()</b> and <b>printAVSL()</b></td></tr>
<tr><td><i>void</i></td><td>printFragmentList(caption)</td><td>Outputs information for each fragment in the AVSL. The output format is:

```
caption
AVSL start 0x######## stop 0x########
Fragment   #  low Water  high Water   size
cells
 Fragment ### 0x######## 0x########   #######
######
```

</td></tr>
</table>

Table 3.5-1 Debug Methods

return	function()	Description
void	printState(string)	String is the caption. Outputs the **getSlipState()** data.
void	sysInfo(ostream&)	Output system information, size of system objects.

Well, some background is probably needed to make sense out of what is being offerred. **getSlipState()**, **printClassSizes()**, and **printState()** can probably stand up without further discussion. For the rest, SLIP operation is discussed below.

Each SLIP cell, SlipHeader, SlipSublist, SlipDatum, and SlipReaderCell, is the same size. The AVSL is a storage place for unallocated cells, and the AVSL space is managed by the SLIP system. The fact that each cell is the same size means that there is no requirement for a garbage collector, all 'holes' in the AVSL space are the same size and it is unneeded to perform space compression and reorganization because of unequal sized space fragments. Remember that the application data types are controlled by the application, and any fragmentation and garbage collection issues are the application's responsibility.

The AVSL space is retrieved from the C++ heap in fragments. The first fragment is allocated either using a default amount or by explicit application choice, and each additional fragment size is determined in the same way, by default or by application choice. The meta-data supporting space allocation is returned in **getSlipState()**.

As cells are used by the application the AVSL space is diminished. When the total available space becomes zero, the next cell allocation request causes an additional fragment of memory to be retrieved from the heap. This increases the total AVSL space (both used and unused cells) by the number of cells in the fragment.

The process of allocation from the heap continues until the C++ kernel reports that no more space is available, that is, the space available in the heap is insufficient to satisfy the next fragment request. This triggers a life threatening and fatal error message, and SLIP quits.

Now, as each fragment is allocated, the cells in the fragment are included in the AVSL. However, the requests for cells from the AVSL and the return of cells to the AVSL is random. In time, the strict relationship between the address of a cell in a fragment is lost, and the ordinal position of the cell in the AVSL is dominant. However, since a cells address in memory doesn't change this lets us view space in two ways. In one, we are concerned with where a cell exists in the AVSL. This yields random cell addresses depending on the ordinal position of a cell in the AVSL. In the other we can treat memory as a series of contiguous addresses, each contiguous region corresponding to a memory fragment. In this way we can output all memory by ordinal position in the AVSL, **printAVSL()**, or by ordinal position in contiguous regions of memory, **printMemory()** and **printFragmentList().**

Before we continue, and just as a side note, as cells are restored to the AVSL there is no effort to compress memory by removing unneeded fragments, or to perform a garbage collection function to reorganize cells in the AVSL so that fragments can be restored to the heap. The AVSL grows but is never reduced. An application using SLIP should keep in mind that SLIP memory utilization grows to a high-water mark and then stays there. Momentary fluctuations in memory needs are permanent.

The printing of cells is in hexadecimal. The cell address, the left link and right link, and other values are output as hexadecimal values. For SlipDatum cells, the output is a combination of hexadecimal values for the SLIP administration values, and if possible, a configured string for data,. That is, if the data value is a pointer, then the pointer value is output in hexadecimal. If the pointer has a sensible string value, then this value is formatted and output also. If the data value is a number or boolean then both the hexadecimal value for the number or boolean and the string representation of the number of boolean are output.

printAVSL() outputs the list of unallocated cells as they are stored in the AVSL. The first cell output is the first cell in the AVSL, and the last cell is the last cell in the AVSL. The physical address of the calls, whether they are from any given fragment, is random, depending on the sequence of cell allocation from the AVSL and deallocation to the AVSL.

There are certain relationships that SLIP cells have that must be satisfied. The relations are checked and warnings issued during output by **printAVSL()**. Their meanings are:

- Left link is not 0xdeadbeef or 0xdeadbeefdeadbeef: Cells in the AVSL (unallocated cells) are singly linked. The left link is set to **0xdeadbeef**. If this is not true then SLIP has failed.
- All SlipSublist objects must reference a SlipHeader. When a SlipSublist object is deallocated to the AVSL, this reference must be present. As a note, when an allocation request is made for a SLIP cell, if the cell is a SlipSublist object, at that time the reference count of the referenced SlipHeader is decremented. When the SlipHeader reference count is zero, the SlipHeader and its list are deallocated.
- The Sublist header pointer does not point to a header. By definition a SlipSublist must contain a reference to a SlipHeader object. If the object referenced is not a SlipHeader object then something is definitely wrong.
- Header reference count > 0. Header is still active. If a SlipHeader object is in the AVSL (the left link is **0xdeadbeef**) then the reference count must be zero. If the reference count is not zero then this indicates that there are live references to the SlipHeader object. This is a problem.

- Number of cells in AVSL does not agree with free space count Descending into sublists is not done. The output cells correspond to the cells immediately available for allocation. SlipSublists are unevaluated to determine if the contained list referenced by the sublist should be be deallocated. The 'avail' field in **getSlipState()** is this number. If the actual count is not the same then there is more (less) space on the AVSL then the counting of allocation/deallocation events.

Not that fingers should be pointed, but some errors are most likely due to the application misusing cell addresses. With the ability to retrieve a cell pointer using getRightLink() and getLeftLink(), there is the ability to modify the cell directly. If not done correctly this will destroy SLIP integrity. As a caution, there are methods available to modify all SLIP cell fields' needed by an application. If a method is not available that permits a modification then this most probably means that the application should not change it. If there is a method that permits an application wanted modification, then probably this method should be used. Most modification methods are inline methods, and there is no performance penalty for using them. Be afraid.

avslHistory() outputs each AVSL allocation/deallocation event. The history can be turned on or off at the applications discretion. Although the AVSL has a defined and limited total space, **avslHistory()** does not. It can continue forever. Consider an allocation of space for 10 cells. If a loop is created where a SlipDatum object is alternatively allocated and deallocated, then the total number of events output depends on the iteration count of the loop. The history is useful in detecting errors of use but the output can be very large.

In summary what we have are two ways of looking at memory. Either as contiguous addresses in a heap allocated fragment, or as a non-contiguous collection of cells on the AVSL. The total space given when we retrieve the SLIP state reports the sum of all the space in all the fragments. The available space given when we retrieve the SLIP state does not report the space which would be available if SlipSublist referenced lists were deallocated. Modifying SLIP cells directly can lead to a failure of SLIP.

3.6 Miscellaneous

Well, there are always that oddball collection of methods which don't easily fit into any particular category and which are too small in scope to present as a separate section. Here they are in Table 3.6-1 Miscellaneous Methods. To call each method except **slipInit()** use **X.method(),** where **X** is an object of any class derived from SlipCellBase. **slipInit()** is called using the syntax, **SlipCellBase::slipinit()**.

Table 3.6-1 Miscellaneous Methods

<table>
<tr><th>return</th><th>((SlipCell&)X).method()</th><th>Description</th></tr>
<tr><td>Void</td><td>deleteSlip()</td><td>Frees all allocated memory. A slipInit() must be called to if SLIP usage is required.</td></tr>
<tr><td>string</td><td>dump()</td><td>Hexadecimal dump of a cell.</td></tr>
<tr><td>string</td><td>dumpLink()</td><td>Hexadecimal dump of the previous/next pointers in a cell</td></tr>
<tr><td>ClassType</td><td>getClassType()</td><td>Each SLIP cell has a unique enumerated type:
<table>
<tr><th>enumeration</th><th>description</th></tr>
<tr><td>eUNDEFINED</td><td>Unallocated AVSL cell</td></tr>
<tr><td>eBOOL</td><td>SlipDatum <i>bool</i> object</td></tr>
<tr><td>eDOUBLE</td><td>SlipDatum <i>double</i> object</td></tr>
<tr><td>eHEADER</td><td>SlipHeader object</td></tr>
<tr><td>eLONG</td><td>SlipDatum <i>long</i> object</td></tr>
<tr><td>ePTR</td><td>SlipDatum <i>application</i> defined object</td></tr>
<tr><td>eREADER</td><td>SlipReaderCell – not the SlipReader</td></tr>
<tr><td>eSTRING</td><td>SlipDatum <i>string</i> object</td></tr>
<tr><td>eSUBLIST</td><td>SlipSublist</td></tr>
<tr><td>eUCHAR</td><td>SlipDatum <i>unsigned</i> char object</td></tr>
<tr><td>eLONG</td><td>SlipDatum <i>long</i> object</td></tr>
</table></td></tr>
<tr><td>string</td><td>getName()</td><td>UTF-8 string representation of the cell object type.</td></tr>
<tr><td>void</td><td>slipInit(alloc, delta)</td><td>Sets the initial memory allocation size and the incremental allocation size. This method should be called before the first SLIP operation is executed. The default values of 10,000 initial SLIP cells and 10,000 additional SLIP cells are used as the allocation amounts. If the delta = 0 then no new space is allocated.</td></tr>
<tr><td>string</td><td>toString()</td><td>Pretty-printed SlipDatum value, hexadecimal dump of other SLIP cells.</td></tr>
</table>

4 .0 SlipCells

This section includes all SLIP objects which can be a list cell. No other object than those defined here can be in a list. All of the classes defined in this section can use the methods in Table 4-1 SlipCell Methods. Each method is called by **X.method()** where **X** is a SlipCell object.

Table 4-1 SlipCell Methods

return	((SlipCell&)X).method()	Description
SlipCell*	getLeftLink()	Pointer to the previous cell in a list.
SlipCell*	getRightLink()	Pointer to the next cell in a list.
SlipCell&	unlink()	Remove the current SLIP cell from a list. This is not a delete. The list is restructured around the removed cell. The SlipHeader can not be unlinked. Unless the unlinked cell is placed into another list, it must be explicitly deleted.

4.1 SlipHeader

Every list has one and only one unique header. The list header is responsible for remembering the first and last cells on the list, tracking the number of lists the current list is a shared in, providing the application the ability to 'mark' the list, and providing an associative list (called the Descriptor List) for application use.

A list header and each cell in the list must be from the AVSL. No list and no list cell can be from the heap or the runtime stack or from any application data store. The list must be created using **new**, as in SlipHeader& list = ***new** SlipHeader();

Once a list is created, the application is responsible for deleting the list. If the list becomes a member of another list this initial deletion will not cause the list to be destroyed. If the list is not a member of another list then the list header and all list cells will be returned to the AVSL. When the final list that the current list is a shared with is deleted, the current list will be deleted. That is, the current list must be deleted once. If the current list is not subordinate to another list, then this initial deletion will delete the list and its contents to the AVSL. If the current list is shared with other lists, then deletion of the current list is automatic when all the other lists are deleted.

Deletion of a list must use the **deleteList()** method and not the operator *delete*. Using the operator *delete* causes the list to be inaccessible in all lists that it is a member of (and a diagnostic message and exception will be generated).

Table 4.1-1 SlipHeader Constructors and Destructors gives the creation and deletion methods for a SlipHeader object.

Table 4.1-1 SlipHeader Constructors and Destructors

return	Format	Description
SlipHeader*	new SlipHeader()	Create a SlipHeader from the AVSL
SlipHeader*	new SlipHeader(SlipHeader&)	Create a SlipHeader from the AVSL and populate it with a copy of the cells from the input list
void	((SlipHeader&)X).deleteList()	Reduces the reference count by 1. If the reference count is zero, return the list and the SlipHeader to the AVSL

Example 4.1.1-1 shows what happens when a list is created, used, and deleted.

```
SlipHeader L1 = new SlipHeader(); // L1: ()
SlipHeader L2 = new SlipHeader(); // L2: ()
L2->insLeft(L1);    // L2: ( () )
L1.deleteList();    // L1: Still exists
L2.deleteList();    // L1: kaput, L2 kaput
```

Example 4.1-1 SlipHeader Delete

The reference count of L1 and L2 is set to '**1**' during initial creation (reference counts are explained below). During the insert of L1 into L2, the reference count of L1 is incremented. When L1 is deleted, the reference count is decremented but since it is not zero, the list is not recovered to the AVSL, i.e., it still exists. When L2 is deleted its reference count is decremented causing it to be recovered to the AVSL. When the AVSL uses the SlipSublist reference to L1, it decrements the reference count, and since if it is zero, ir will then recover L1 to the AVSL. There is one little fly stuck to the fly paper. The C++ variables., L1 and L2, remain pointing to a non-existent list. This is an artifact of being an API instead of a system. This condition can be checked by using various interrogatories until L1 or L2 is reused. At that time the variables will point to an object whose current use is valid but which is not the same as the application assumes. Or, if your not careful your dead.

The following constructor uses are illegal:

```
SlipHeader L;             // Attempt to create a list header on the stack
SlipHeader L();           // Didn't learn the first time
SlipHeader L1(L2);        // Attempt to repeat your error & copy L2
```

SlipHeader objects contain an associative list. The associative list is a 2-tuple, <***key, value***> pair, where the **key** is a searchable item and **value** it's associated value. The section on Description Lists will address this more fully.

4.1.1 SlipHeader Methods

SlipHeader has some characteristics unique to its nature and useful to applications. An application can empty a list, **flush(),** query the list to see if it is empty, **isEmpty()**, place an application mark into the list and then query a list for its mark, **putMark()**, **getMark()** respectively.

A list has a size (the number of cells in a list) and an application can get this size, **size()**. The size is defined as the number of cells visible in the list without descending into sublists. It is not a recursive search of the list to determine the total size of the topmost list and all subordinate lists. A note, getting the size is slow.

Each SlipHeader object maintains a reference count giving the number of times the list is referenced. When the list is created, this reference count's value is set to '1'. Each time a sublist references the list, the reference count is incremented, and each time that a sublist is deleted (and hence, the reference is removed) the reference count is decremented. The total outstanding references to a list can be no more than 65,535. **getRefCount()** gives the application the ability to see the current number of references to a list.

There is a 15-bit field set aside for exclusive use by an applcation. There are no restrictions on use of this field other than it's size. Access to the data is given with **getMark()** and storage of data into the field is given by **putMark()**.

In a manner similar to getting a substring, it is possible to create a new list by removing a fragment of a more established one. In **splitLeft()**, all list cells to the left of and including the input list are used to create a new list. The list cells are removed from the current list and moved to a list just created. In the same way, **splitRight()** creates a new list and moves all list cells to the right of and including the current list the new list. If the input cell is the list header, the resulting new list is empty;

There is a close relationship between a SlipHeader, the list container, and a SlipSublist object, a reference to a list. In many cases operations which involve a SlipHeader and a SlipSublist will use the SlipHeader reference in the SlipSublist object. The descriptive text or context in tables and figures will make clear when the SlipSublist requires a the SlipSublist object or the the SlipHeader reference contained in a SlipSublist.

Many of the common operations have a special nomenclature when the SlipHeader is the source or destination. The nomenclature makes used of standard views of a list and standard ways of addressing such use. In each case the invoking method is called with **(**(SlipHeader)**X).method()**. The original, common, methods are still available for use and are interchangeable with their SlipHeader synonyms. A summary list of nomenclature changes is given below.

((SlipHeader)X).method()

common name	**new name**	**Description**
insRight()	push()	Insert an object to the top of a list (before the first cell).
InsLeft()	enqueue()	Insert an object to the bottom of a list (after the last cell).
replace()	replaceBot()	Replace the last object in a list
replace()	replaceTop()	Replace the first object in a list.
unlink()	dequeue()	Unlink the last cell in a list and return it's reference.
unlink()	pop()	Unlink the first cell in a list and return i'ts reference.

A complete list of SlipHeader general methods are provided in Table 4.1.1-1 SlipHeader General Methods. All methods are invoked by **(**(SlipHeader&)**X).method().**

Table 4.1.1-1 SlipHeader General Methods

return	**((SlipHeader&)X).method()**	**Description**
SlipCell&	dequeue()	Unlink from the list tail and return it's reference
SlipHeader&	enqueue(SlipHeader&)	Insert a **new** SlipSublist reference to the SlipHeader to the list tail and return a reference to the header

Table 4.1.1-1 SlipHeader General Methods

return	((SlipHeader&)X).method()	Description
SlipHeader&	enqueue(SlipSublist&)	Insert a copy of the SlipSublist to the list tail and return a reference to the header
SlipHeader&	enqueue(SlipDatum&)	Insert a copy of the SlipDatum to the list tail and return a reference to the header
SlipHeader&	enqueue(*bool*)	Insert a **new** SlipDatum *bool* to the list tail and return return a reference to the header
SlipHeader&	enqueue(*char*)	Insert a **new** SlipDatum *char* to the list tail and return a reference to the header
SlipHeader&	enqueue(*unsigned char*)	Insert a **new** SlipDatum *unsigned char* to the list tail and return a reference to the header
SlipHeader&	enqueue(*long*)	Insert a **new** SlipDatum *long* to the list tail and return a reference to the header
SlipHeader&	enqueue(*unsigned long*)	Insert a **new** SlipDatum *unsigned long* to the list tail and return a reference to the header
SlipHeader&	enqueue(*double*)	Insert a **new** SlipDatum *double* to the list tail and return a reference to the header
SlipHeader&	enqueue(*PTR, const* void* operation=**default**))	Insert a **new** SlipDatum *PTR* to the list tail and return a reference to the header (see User Defined Types)
SlipHeader&	enqueue(*string&*, *bool c*onstFlag=**false**)	Insert a **new** SlipDatum *string* to the list tail and return a reference to the header (see User Defined Types)
SlipHeader&	enqueue(*string**, *bool* constFlag=**false***)*	Insert a **new** SlipDatum *string* to the list tail and return a reference to the header (see User Defined Types)
SlipCell&	getBot()	Return a reference to the list last cell
uShort	getMark()	Return the 15-bit application field value
uShort	getRefCount()	Return the number of list references
SlipCell&	getTop()	Return a reference to the list first cell
SlipHeader&	flush()	Empty the list
SlipCell&	pop()	

Table 4.1.1-1 SlipHeader General Methods

return	((SlipHeader&)X).method()	Description
SlipHeader&	push(SlipHeader&)	Insert a **new** SlipSublist reference to the SlipHeader to the list front and return a reference to new cell
SlipHeader&	push(SlipSublist&)	Insert a copy of the SlipSublist to the list front and return a reference to the header
SlipHeader&	push(SlipDatum&)	Insert a copy of the SlipDatum to the list front and return a reference to the header
SlipHeader&	push(*bool*)	Insert a **new** SlipDatum *bool* to the list front and return a reference to the header
SlipHeader&	push(*char*)	Insert a **new** SlipDatum *char* to the list front and return a reference to the header
SlipHeader&	push(*unsigned char*)	Insert a **new** SlipDatum *unsigned char* to the list front and return a reference to the header
SlipHeader&	push(*long*)	Insert a **new** SlipDatum *long* to the list front and return a reference to the header
SlipHeader&	push(*unsigned long*)	Insert a **new** SlipDatum *unsigned long* to the list front and return a reference to the header
SlipHeader&	push(*double*)	Insert a **new** SlipDatum *double* to the list front and return a reference to the header
SlipHeader&	push(*PTR, const* void* operation=**default**))	Insert a **new** SlipDatum *PTR* to the list front and return a reference to the header (see User Defined Types)
SlipHeader&	push(*string&*, *bool c*onstFlag=**false**)	Insert a **new** SlipDatum *string* to the list front and return a reference to the header (see User Defined Types)
SlipHeader&	push(*string**, *bool* constFlag=**false***)*	Insert a **new** SlipDatum *string* to the list front and return a reference to the header (see User Defined Types)
uShort	putMark()	Put a 15-bit value into the lislt mark field
SlipHeader&	replace(const SlipHeader&)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(const SlipSublist&)	Illegal. SlipHeader can not be replaced

Table 4.1.1-1 SlipHeader General Methods

return	((SlipHeader&)X).method()	Description
SlipHeader&	replace(const SlipDatum&)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(*bool)*	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(*char*)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(*unsigned char*)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(*long*)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(*unsigned long*)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(*double*)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(const *PTR*, const void* operation=**null**)	Illegal. SlipHeader can not be replaced
SlipHeader&	replace(const *string*&, bool constFlag=**false**)	Illegal. SlipHeader can not be replaced
SlipSublist&	replaceBot(const SlipHeader&)	Replaces the list tail with a **new** SlipSublist object
SlipSublist&	replaceBot(const SlipSublist&)	Replaces the list tail with a **new** SlipSublist object
SlipDatum&	replaceBot(const SlipDatum&)	Replaces the list tail with a copy of the input
SlipDatum&	replaceBot(*bool)*	Replaces the list tail with a **new** SlipDatum *bool*
SlipDatum&	replaceBot(*char*)	Replaces the list tail with a **new** SlipDatum *char*
SlipDatum&	replaceBot(*unsigned char*)	Replaces the list tail with a **new** SlipDatum *unsigned char*
SlipDatum&	replaceBot(*long*)	Replaces the list tail with a **new** SlipDatum *long*
SlipDatum&	replaceBot(*unsigned long*)	Replaces the list tail with a **new** SlipDatum *unsigned long*
SlipDatum&	replaceBot(*double*)	Replaces the list tail with a **new** SlipDatum *double*
SlipDatum&	replaceBot(const *PTR*, const void* operation=**null**)	Replaces the list tail with a **new** SlipDatum *PTR*
SlipDatum&	replaceBot(const *string*&, bool constFlag=**false**)	Replaces the list tail with a **new** SlipDatum *string*

Table 4.1.1-1 SlipHeader General Methods

return	((SlipHeader&)X).method()	Description
SlipDatum&	replaceBot(const *string**, bool constFlag=**false**)	Replaces the list tail with a **new** SlipDatum *string*
SlipSublist&	replaceTop(const SlipHeader&)	Replaces the list front with a **new** SlipSublist object
SlipSublist&	replaceTop(const SlipSublist&)	Replaces the list front with a **new** SlipSublist object
SlipDatum&	replaceTop(const SlipDatum&)	Replaces the list front with a copy of the input
SlipDatum&	replaceTop(*bool)*	Replaces the list front with a **new** SlipDatum *bool*
SlipDatum&	replaceTop(*char*)	Replaces the list front with a **new** SlipDatum *char*
SlipDatum&	replaceTop(*unsigned char*)	Replaces the list front with a **new** SlipDatum *unsigned char*
SlipDatum&	replaceTop(*long*)	Replaces the list front with a **new** SlipDatum *long*
SlipDatum&	replaceTop(*unsigned long*)	Replaces the list front with a **new** SlipDatum *unsigned long*
SlipDatum&	replaceTop(*double*)	Replaces the list front with a **new** SlipDatum *double*
SlipDatum&	replaceTop(const *PTR*, const void* operation=**null**)	Replaces the list front with a **new** SlipDatum *PTR*
SlipDatum&	replaceTop(const *string*&, bool constFlag=**false**)	Replaces the list front with a **new** SlipDatum *string*
SlipDatum&	replaceTop(const *string**, bool constFlag=**false**)	Replaces the list front with a **new** SlipDatum *string*
unsigned	size()	Number of cell in topmost list w/o SlipHeader
SlipHeader&	splitLeft(SlipCell&)	New list with cells to the left including the input cell
SlipHeader&	splitRight(SlipCell&)	New list with cells to the right including the input cell

The relational operators equal (**==**) and not equal (**!=**) are used to compare a SlipHeader object with another SlipHeader object or SlipSublst object. The result of the comparison is **true** when the SlipHeader reference refer to the same object. Any other data type other than a SlipHeader or SlipSublist will return **false**. The remaining relational operations (<. <=. >=, >) will return **false**. There is no sensible relationship between references other than equality. Relational operations are commutative, **X == Y** and **Y == X** will yield the same result.

Table 4.1.1-2 SlipHeader Relational Operators shows the results of using **==** and **!=**. All other relational operators yield **false**.

Table 4.1.1-2 SlipHeader Relational Operators

return	SlipHeader&	RHS	Description
bool	X	== (SlipHeader&)Y	Comparison legal
bool	X	== (SlipSublist&)Y	Comparison legal
false	X	== (SlipDatum&)Y	Comparison illegal
false	X	== literal	Literal values are not legal
bool	X	!= (SlipHeader&)Y	Comparison legal
bool	X	!= (SlipSublist&)Y	Comparison legal
false	X	!= (SlipDatum&)Y	Comparison illegal
false	X	!= literal	Literal values are not legal

There are several ways of determining whether two list references are equal (as opposed to referring to the same list). The references can refer to the same list, they are equal because they are the same. Two different lists can have the same taxonomy, they are structurally equal. Two lists can have the same taxonomy and the data type and values for equivalent locations on the lists are the same, they are structurally and semantically identical. SLIP implements referential and structural equality. The relational imperative, **==,** checks for referential equality. The interrogatory, **isEqual()**, checks for structural equality. It is always true that if two list references are referentially equal then they are structurally equal. It is not always true that if two lists are structurally equal they are referentially equal.

All the common interrogatories are valid with a SlipHeader object, Table 4.1.1-3 Interrogatories, shows those unique to SlipHeader objects.

Table 4.1.1-3 Interrogatories

return	**((SlipHeader&)X).method()**	**Description**
bool	isEmpty()	**true** if the list is empty
bool	isEqual(SlipHeader&)	**true** if two lists are structurally identical

An application can pretty-print the list with **writeQuick()** and **writeToString()** and can create a list file with **write()**. These methods are described in the I/O section. The SlipHeader object can be dumped, dump(), and the list can be dumped. **dumpList()**. The hexadecimal dump outputs the binary for all SlipHeader objects and all SlipDatum and SlipSublist objects in the top level list and all subordinate lists. These methods are given in Table 4.1.1-4 Miscellaneous.

Table 4.1.1-4 Miscellaneous

return	**((SlipHeader&)X).method()**	**Description**
string	dump()	Return a hexadecimal dump of the SlipHeader object
void	dumpList()	Hexadecimal dump of the list to standard output
void	write()	Create a list file and output to cout
void	write(ostream&)	Create a list file and output to a stream
void	write(string&)	Create a list file and output to a file
void	writeQuick()	Output a list definition to cout
void	writeQuick(ostream&)	Output a list definition to a stream
string	writeToString()	Return a list definition

A hastily constructed example in Example 4.1.1-1 SlipHeader.

```
SlipHeader* L1 = new SlipHeader();      // create (1 2 (3 4) 5)
SlipHeader* L2 = new SlipHeader();      // create (5 (4 3) 2 1 (4 3))
SlipHeader* S1 = new SlipHeader();      // create (3 4)
SlipHeader* S2 = new SlipHeader();      // create (4 3)
long one          = 1;
unsigned long two  = 2;
char three         = 3;
unsigned char four = 4;
//------------------------- creation -------------------------//
S1->enqueue((char)3).enqueue (four);       // (3 4)
S2->push(three);                           // (3)
S2->push((long)4);                         // (4 3)
L1->enqueue (one).enqueue(two).enqueue(S1);// (1 2 (3 4)
L1->enqueue ((long)5);                     // (1 2 (3 4) 5)
S2->L2->push((long)5);                     // (5 (4 3) 2 1)
//---------------------- interrogatories -----------------------//
if (L1 == L2)        cout << “equal”   << endl;// no output
if (L1->isEqual(L2)) cout << “isEqual” << endl;// output is “Equal”
if (L1->isEmpty())   cout << “isEmpty” << endl;// no output
//---------------------------- code ----------------------------//
L2->enqueue(S2);                           // (5 (4 3) 2 1 (4 3))
S2->getRefCount()                          // refCount = 3
L2->deleteList();                          // list restored to AVSL
S2->getRefCount();                         // refCount = 3
L1->setMark(0xF1);                         // mark = 0x71
L1->getMark();                             // mark = 0x71
delete &L1->pop();                         // L1: (2 (3 4) 5)
delete &L1->dequeue();                     // L1: (2 (3 4))
L1->replaceBot(S2);                        // L1: (2 (4 3) )
S1->getRefCount();                         // refCount = 2
S2->getRefCount();                         // refCount = 4
L1->replaceTop(“able”);                    // L1: (able (4 3))
//------------------------- destruction -------------------------//
L1->deleteList();                          // list restored to AVSL
S1->deleteList();                          // list restored to AVSL
S2->deleteList();                          // list restored to AVSL
```

Example 4.1.1-1 SlipHeader

Well, commenting on the example:

- Creation is no more than **insLeft()** and **insRight()** replaced by **enqueue()** and **push()** respectively.
- Conditionals check existence features:
  - L1 and L2 do not reference the same list. The conditional fails.
  - L1 and L2 are structurally equal. The conditional succeeds.
  - L1 is not empty. The conditional fails.
- The behavior of **enqueue()** is the same as **insLeft()**. When we enqueue a list header, a SlipSublist object is created referencing the input list.

- Deletion of lists is subject to deferred deletion of cells. When a list is deleted the expected behavior is that the SlipSublist cells will not be recovered from the AVSL until they are needed. When they are needed, the reference count field of the SlipHeader object referenced is decremented. In this case, the **getRefCount()** method is called immediately after the list is deleted with **deleteList()**. No demand is placed on the list cells and the SlipSublist reference has not yet been recovered, hence, the list (S2) reference count is unaltered.
- **setMark()** quietly removes the most significant bit of the input mark. Only 15-bits are allowed and the most significant bit is owned by SLIP. **getMark()** returns a 15-bit value.
- **pop()** removes the topmost list cell and **dequeue()** removes the bottom list cell. The return values are a reference to the removed cell. Note that the cell is not deleted by SLIP.
- **replaceBot()** and **replaceTop()** apply a cell replace() to the bottom or top list cells respectively. The object replaced is deleted to the AVSL.
- **S1->getRefCount()** and **S2->getRefCount()** return the current reference count in the list header cell. This reference count is changed immediately when the SlipHeader list is deleted with **deleteList()**, but changet is deferred for SlipSublist references until the SlipSublist cell in the AVSL is needed. We have made the assumption the there are enough cells in the AVSL so that for this example, the deleted SlipSublist references are not immediately needed. Therefore, the reference counts will be the same reference counts as if all SlipSublist objects were still active.
- **replaceTop("*able*")** shows the simplest form for creation of a SlipDatum cell containing a string.

4.1.2 Description List

A Description List is an optional association list which can be assigned to a SlipHeader object. The association is a 2-tuple containing a key and a value, **<key, value>**. The semantics of a search is that when a key is found, its associated value is returned.

Operations performed on the list are not reflected in the Descriptor List, except for list deletion, **deleteList()**. If a **flush()** is executed, the list cells are deleted but the Descriptor List is unchanged. This applies to all other operations including insertions, cell deletions, replacement, and moves. The list and the Descriptor List it contains are independent.

This associative pair allows information to be stored concerning the list and avoid placing this information in a list. The support methods provide the mechanism for inserting, searching, modifying and deleting tuples.

Both the key and the value can be a SlipHeader. If the application is an insertion, then a SlipSublist is created referencing the SlipHeader and inserted in place of the SlipHeader. If a search is made with a SlipHeader, then the '==' operator is used. Any search operations compare the reference contained in a SlipSublist with the input SlipHeader.

To store a literal into the list as either key or value, it must be stored as a SlipDatum object.

All SlipCell objects stored into the Description List must be from the AVSL. Keys and values can not be temporary or from the heap. They must be from the AVSL, and they will be retained until the list containing the Descriptor List is deleted or until the Descriptor list is deleted or flushed or the applications deletes the <**key**, **value**> pair..

No SLIP checks are made to ensure that there is a relationship between the key and value. They can both be the same and either one or both can be list references (SlipSublists) or SlipDatum objects. SLIP just doesn't care.

Before operations on a Descriptor List are begun, the list must be created with **create_dList()**. Interrogatories allow searching the list for a key, **containsKey()**, or a value, **contains()**, and for retrieving the associated value for a key, **get()**. The Descriptor List can be deleted, **delete_dList()**, flushed, **flush_dList()**, printed, **printDlist()**, and dumped, **dumpDlist()**. Tuples can be deleted, **deleteAttribute()** and the list size can be found, **size_dList()**. These operations do not affect the list containing the Descriptor List.

If a Descriptor List is deleted, **delete_dList()**, then a new list must be created, **create_dList()** before any other operations on the list can be performed.

Values within a tuple must be deleted with **deleteAttribute()** and recreated with **put()** in order to change the attribute value.

Calling a method uses **(**(SlipHeader&)**X).method()**. Interrogatories always succeed and return **true** or **false**.

The available methods are given in Table 4.1.2-1 Descriptor List Methods.

Table 4.1.2-1 Descriptor List Methods

return	((SlipHeader&)X).method()	Description
bool	contains(SlipCell&)	**true** if the input value is found.
bool	constainsKey(SlipCell&)	**true** if the input key is found.
SlipHeader&	create_dList()	Create a Description List for the SlipHeader.
bool	deleteAttributes(SlipCell&)	Delete a tuple for the input key.

Table 4.1.2-1 Descriptor List Methods

return	((SlipHeader&)X).method()	Description
SlipHeader&	delete_dList()	Delete a Description List. Equivalent to **deleteList()**.
SlipHeader&	flush_dList()	Empty all tuples in the Description List.
SlipCell&	get(SlipCell&)	Return an associated value for the input key or the key.
void	printDList()	Pretty-print the Description List.
void	dumpDList()	Uglify the Description List in hexadecimal.
SlipCell&	put(SlipCell&,SlipCell&)	Put a **<key, value>** tuple into the Description List.
unsigned	size_dList()	Return the number of tuples in the Description List.

4.2 SlipSublist

A SlipSublist object is a list cell which provides a reference to a **SlipHeader** object. This cell is traversable using the Slip iterators, **SlipReader** and **SlipSequencer**, and with some additional difficulty, by explicit application code.

A SlipSublist object allows lists to be referenced within lists. It provides the basic functionality for constructing networks, acyclic graphs, and trees.

A **SlipSublist** object can not be created on the the heap or runtime stack. A **SlipSublist** object not created from the AVSL can not be inserted into a list and can not be deleted. An attempt to create a non-AVSL **SlipSublist** will be detected and cause application termination.

If a **SlipSublist** object is created using **new** then it must be deleted. If the object is put into a list then when the list is deleted the object is deleted. If the object is not put into a list then the application must delete the object before the scope containing the object is exited.

A **SlipSublist** object must be created referencing a list (**SlipHeader** object), as in **new SlipSublist(**list reference**)** or **SlipSublist(**sublist reference**)** . At no time during processing can the SlipHeader reference be **null** nor can it ever reference to an object which is not a list. Operational errors which can create a violation to this condition are given below.

The valid constructors and destructors of a **SlipSublist** object are given in Table 4.2-1 SlipSublist Constructors and Destructors. Note that if the object is inserted into a list, then deletion is implicit when the list is deleted.

Table 4.2-1 SlipSublist Constructors and Destructors

result	format	Description
SlipSublist*	**new** SlipSublist(SlipSublist&)	Create an object from the AVSL.
SlipSublist*	**new** SlipSublist(SlipHeader&)	Create an object from the AVSL.
void	**delete** (SlipSublist*)cell	Return an object to the AVSL

If you delete the **SlipSublist** object, then the object is deleted and a **deleteList()** is executed against the referenced header. If you delete the header instead of deleting the **SlipSublist** object, and if the object is the only reference to the header, then the list will be returned to the AVSL and the **SlipSublist** object will be out of synchrony. Any of the following conditions will (not may) occur:

- The **SlipSublist** header reference is stale and references a cell in the AVSL.
- If the pointer is stale, then iterators (or iteration) will continue in the AVSL, or
- If the deleted header cell is used to create another **SlipSublist** or **SlipDatum** object (operator **new**) then iterators (or iteration) will be to objects on another list.
- If the header cell from the AVSL is used to create another **SlipHeader** object (operator **new**) then iterators (or iteration) is to cells on another list and this may be undetectable.
- Ill effects are latent and may not be 'seen' for some time after the initial **deleteList()**.

The application is guaranteed inexplicable and hard to track errors.

In all other respects, a **SlipSublist** object is a **SlipCell**. It can be moved, inserted, and assigned to, and replaced.

4.2.1 SlipSublist – SlipHeader Methods

The **SlipSublist** provides access to all **SlipHeader** operations which are not in conflict with **SlipSublist** operations. The conflict is defined as all operations which can have a legitimate meaning for both **SlipHeader** objects and **SlipSublist** objects. In the case of a conflict, the interpretation is that the operation applies to **SlipSublist** objects and not **SlipHeader** objects. All **replace()**, **print()**, and **dump()** methods apply to the SlipSublist object and not the referenced SlipHeader object**.**

The refCnt field of the header is decremented on execution of a **deleteList().** If this count falls to zero then the header and all list cells are restored to the AVSL. If the refCnt is not zero then at some unknown future time when a **delete** *(*SlipSublist&*)object* or

(SlipHeader&)header.**deleteList()** the header and list cells will mysteriously disappear onto the AVSL.

There is a subtle difference between a SlipHeader and a SlipSublist. A SlipHeader names a list and a SlipSublist refers to that name. A SlipHeader is a member of it's own list but retains its identity and can not be a member of any other list. A SlipSublist contains a reference to a list. A SlipSublist allows a list to be a member of other lists.

All lists must be acyclic. This means that in any path starting at a SlipHeader object there can be no SlipSublist referring to this header. If a lists is not acyclic then the list can not be deleted and restored to the AVSL. All memory included in the top level list and all sublists (the complete graph with initial node of the given object) will be lost for 40 years in the desert.

The SlipHeader methods directly usable by a SlipSublist object are given below. Details on the methods are given in the SlipHeader section and are not repeated here.

((SlipSublist&)X).method()	
dequeue()	Removal of the bottom cell from a list
dumpList()	Ugly hexadecimal of a list and all sublists.
enqueue()	Insert a cell to the bottom of a list.
flush()	Empty a list.
getBot()	Peek at the last cell in a list.
getTop()	Peek at the first cell in a list.
getMark()	Look at the application mark of a list.
getRefCount()	Look at the list membership count.
isEmpty()	**true** if the list is empty
isEqual()	**true** if two list are structurally identical.
pop()	Remove the top cell in a list.
printList()	Pretty-print the list.
push()	Insert a cell to the top of a list.
putMark()	Set an application mark into the list.
replaceTop()	Replace the top cell of a list.
replaceBot()	Replace the bottom cell of a list.

size()	Return the number of cells in the list top.
SplitLeft()	Create a new list from the list top to the current cell.
SplitRight()	Create a new list from the current cell to the list bottom.

The Descriptor List methods directly usable by a SlipSublist object are given below. Details on the methods are given in the SlipHeader section and are not repeated here.

((SlipSublist&)X).method()

constains()	true if the input value is in the Descriptor List.
containsKey()	**true** if the input key is in the Descriptor List.
create_dList()	Creates a new Descriptor List.
deleteAttributes()	Deletes a **<key, value>** attribute.
delete_dList()	Deletes a Descriptor List.
dumpDlist()	Hexadecimal dump of a Descriptor List
flush_dList()	Remove all **<key, value>** pairs from the list.
get()	Return the value of a **<key, value>** pair.
isDList()	**true** if the list has a Descriptor List.
printDlist()	Pretty-print the Descriptor List.
put()	Put a **<key, value>** pair into a Descriptor List.
size_dList()	Number of **<key, value>** pairs in the Descriptor List.

4.2.2 SlipSublist Methods

SlipSublist supports a limited number unique methods. Assignment is supported if the RHS is either a **SlipSublist** or a **SlipHeader**, and causes the referenced **SlipHeader** in the current **SlipSublist** object to be deleted and the new **SlipHeader** referenced in either another **SlipSublist** object or a **SlipHeader** to replace it after the **SlipHeader** reference count is incremented. All replace operations cause the existing SlipHeader referenced in the SlipSublist object to be deleted and replaced by a new **SlipDatum** object or **SlipHeader** reference, with the reference count of the new list incremented.

Equality checks for exact equality of **SlipHeader** objects reference the current referenced list. If the current **SlipSublist** list reference equals the RHS **SlipHeader** reference then **true** is returned. In all other cases, **false** is returned. **dump()** and **toString()** return internal formats for the **SlipSublist** object, and **getHeader()** returns the **SlipHeader** reference. And as Walter Lanz used to say, “That's all folks”.

Table 3.2-1 Replace Methods shows the result of replacing a **SlipSublist** object.

Table 4.2.2-1 Assignment shows the results of assignment, (**SlipSublist**&)X = RHS, where RHS is any of the recognized SLIP types.

Table 4.2.2-1 Assignment

return	**RHS**	**Description**
SlipSublist&	SlipHeader&	Delete the existing list, increment the refcount of the new list & use.
SlipSublist&	SlipSublist&	Delete the existing list, increment the refcount of the new list & use.
SlipSublist&	SlipDatum	illegal
SlipSublist&	literal	Illegal (DataType = bool, discrete data, string application data)

Table 4.2.2-2 Relational Operators shows the results of using **==** and **!=**. All other relational operators yield **false**.

Table 4.2.2-2 Relational Operators

return	**SlipSublist&**	**RHS**	**Description**
bool	X	== (SlipHeader&)Y	Comparison legal
bool	X	== (SlipSublist&)Y	Comparison legal
false	X	== (SlipDatum&)Y	Comparison illegal
false	X	== literal	Literal values are not legal
bool	X	!= (SlipHeader&)Y	Comparison legal
bool	X	!= (SlipSublist&)Y	Comparison legal
false	X	!= (SlipDatum&)Y	Comparison illegal
false	X	!= literal	Literal values are not legal

Table 4.2.2-3 Miscellaneous Methods shows the miscellaneous operations and their result.

Table 4.2.2-3 Miscellaneous Methods

return	method()	Description
string	dump()	Internal representation of SlipSublist object.
SlipSublist&	getHeader()	Returns the list referenced in the SlipSublist object.
string	toString()	Internal representation of SlipSublist object.

4.3 SlipDatum

The **SlipDatum** cells are **SlipCells** and the objects containing application defined values. The cells participate is cellish activities (**insert()**, **delete**, **replace()**) and values participate in arithmetic, bit, cast, logical, and unary operations.

SlipDatum cells can be created on the runtime stack or from the AVSL. Runtime stack objects can not be inserted in a list. In all other respects, runtime objects are a **SlipDatum** object.

Section 2 Data defines all of the value related operations and defines data types and value ranges. Section 3 Common Operations describes the cell operations that **SlipDatum** cells can participate in.

There is a special **SlipDatum** value for User Data Types defined in Section 4.4 Application Data Types, This allows arbitrary application defined data to be handled in SLIP.

Assignment treats the LHS data type as polymorphic and casts the LHS the the RHS, as in Example 4.3-1 Assignment:

```
SlipDatun X((bool)false);
SlipDatum Y = (bool)true;
SlipDatum Z = (long)15;
SlipDatum a = (double)3.5
X = Y;                      // true
X = Z;                      // true = (bool)Z
Z = a;                      // 3.5
z = Z;                      // 15
```

Example 4.3-1 Assignment

Some noteworthy methods are given below:

	(SlipDatum&)X.method()	
string	dump()	Return a string with the internal rendition of the cell.
bool	isData()	Is this a SlipDatum cell.
bool	isDiscrete()	Is this a SlipDatum cell with a discrete number.
bool	isNumber()	Is this a SlipDatum cell with a number.
bool	isPtr()	Is the cell a application defined SlipDatum cell.
bool	isReal()	Is this a SlipDatum cell with a real.
bool	isString()	Is this a SlipDatum cell for a string.
string	toString()	Return a string with the formatted value in the cell.

Some simple programming examples using SlipDatum object is given in Example 4.3-2:

```
 1: SlipHeader* polyNomial = new SlipHeader();
 2: SlipSequencer iter(polyNomial);
 3: SlipDatum X = (double)1.0;
 4: SlipDatum Y = (double)2.0;
 5: polyNomial.push(Y);
 6: polyNomial.push(X);

 7: double hypotenuse = sqrt(X*X + Y*Y);

 8: double value = 0;

 9: iter.advanceLWR();
10: do while(!iter.isHeader()) {
11:    value += iter.currentCell();
12:    iter.advanceLWR();
13: };

14: double sqrt(SlipDatum& X) {
15:   return ::sqrt((double)X);
16: };

17: polynomially();
```

Example 4.3-2 SlipDatum

The example is simple by design. It does the following:

line #	Description
1	Create a list named polyNomial.
2	Create a simple iterator for the list. The iterator initially references the list header.
3 - 4	Create two (*double*) **SlipDatum** objects.
5 - 6	Insert a copy of X & Y into the list. The list will look like (X Y). A copy is created because X & Y are on the runtime stack and will be deleted when the current scope is exited. Lists require persistent data, and this is achieved by creating a **SlipDatum** object and inserting the value of the runtime **SlipDatum** data type and value into the newly created cell.
7	Calculate the hypotenuse of a right triangle with value X and Y. Note that C++ sqrt must be overridden because it doesn't understand SlipDatum objects.
9 – 13	Treat the list as a polynomial and calculate $\sum a_i x_i$ where a_i are the coefficients in the array and x_i are assumed to be (*double*)1.0. This calculation will only work if all objects are **SlipDatum** cells.
9	The iterator initially references the list header. Advance to the first cell in the list.
10	If the next cell to be evaluated is the list header, we are done.
11	Add $a_i x_i$ to the variable value.
12	Iterate to the next cell in the list.
14 - 16	Override the square root function. The 'new' square root function casts the input **SlipDatum** value to a *double* and returns its square root.
17	The list must be deleted (if it is no longer needed). Since this is the only reference to the list, when the list is deleted then all of the list contents will also be deleted and restored to the AVSL. The actual deletion of the list is not immediate. Outstanding pointers and references to the list header and list cells are undetectably stale.

4.4 Application Data Types

An Application Data object is handled in an identical fashion as any other **SlipDatum** object. It participates in list operations and can support arithmetic, boolean, bit, assignment, logical, casting, and unary operations. List operations are supported as a part of the SLIP interface to a **SlipDatum** cell. Computational operations are associated with the Application Data object when the application **SlipDatum** cell is created. The **SlipDatum** object is a container for the Application Data object.

Definition of Application Data content and behavior is under the control of the application. The application must derive a class from **SlipPointer** and expand the **SlipPointer** defined behavior in an application specific, way including additional methods and data items. This has no effect of SLIP operations. The **SlipPointer** class is an ADT and can not be used directly.

SLIP assumes that the User Data is persistent. When the data is placed into a list, it is assumed that the data will last until the list is deleted. The application must be aware of this and not perform operations to invalidate this. The data must be present as long as the SlipDatum cell containing the data is live. Application Data can not be placed on the stack. It must come from either the heap or from an application static area of memory, or from an application defined space allocator. If the Application Data is constructed on the stack, when the current scope is exited the **SlipDatum** object containing the Application Data will have a stale pointer and who knows what damage will be done (only the Shadow does).

Application Data Type is distinguished from all other objects in SLIP in that memory utilization is under application control and not under SLIP control. The application is responsible for the management of space required to hold the data, for acquiring space and deleting space.

There are two user data types, *strings* and Application Data Types. The *string* semantics are known and defaults are provided. Application Data Type semantics are unknown and the Application is responsible for deriving a subclass from the SlipPointer and SlipPtrOp class.

The **SlipPtrOp**.h class allows the application to override default arithmetic, assignment, bit, boolean, logical and unary operations. This will be covered in the Application Data Type section of this document. The SlipPtrOp object is referenced during these operations allowing an application type participation in operations in a transparent fashion.

For Application Data Types SLIP interfaced memory management is solely contained in the SlipPointer Abstract Data Type (ADT) derived class. The interface contains methods equivalent to **new**, **delete**, and const copying. These operations are used by SLIP when data objects are created, deleted, and copied. Whereas the defaults in the SlipPtrOp class can be used without change, SlipPointer must be inherited and functionality for virtual methods provided. There is no default.

The default constructors/destructors of the base class have very little functionality. The constructor saves the Application Data Type name supplied or defaults to “nullName” and the destructor provides decoration, it has no body. This reflects the lack of assumed knowledge about the characteristics of the data at the base class level. Multiple different Application Data Types should have different, unique, names.

The supplied constructors and destructors are given in Table 4.4-1 SlipPointer Constructors/Destructors.

Table 4.4-1 SlipPointer Constructors/Destructors

method()	Description
SlipPointer(const string* = “nullName”)	Save application data name
~SlipPointer()	Does nothing

There are no defaults for the methods in Table 4.4-2 Application Data Type Operations. These methods must be supplied by the application.

Table 4.4-2 Application Data Type Operations

return	(SlipPointer&)X.method()	Description
SlipPointer*	copy()	Return a copy of the current object.
string	dump()	Internal description of application data
*string**	getName()	Name of the application data
typedef†	getParse()	Returns a pointer to the parse method
SlipDatum&	parse(SlipHeader*)	Factory to create a SlipDatum object
void	remove()	Destructor
string	toString()	Pretty-print application data
string	write()	Return a string suitable for output.

† *typedef SlipDatum& (*Parse)(SlipHeader& head);*

- **copy():** SLIP copies values (and types) during various operations. The application must provide an application specific method to perform this copy and return a pointer to the results. Space needed to create application data is an application responsibility and can not come from the SLIP AVSL. Some hints:
    - Constant: For persistent application data in which the application maintains exclusive control over the deletion of the object and for ensuring that when deletion occurs there are no outstanding instances of the object stored into any **SlipDatum** cell, it is

convenient to return a pointer to a single instance of the object. That is, the object is never explicitly copied.

- Non-Constant:; For data in which the application relinquishes deletion responsibilities to SLIP, i.e., when SLIP deletes a **SlipDatum** object then it is desired that the Application Data object be deleted, then the copy method should create a new object and return a pointer to it.

Note that the copy() and the remove() methods are linked Remove() semantics depends on the decision made for copy.

- **dump():** Application specific data needed for diagnostic purposes. This method is called by SLIP whenever diagnostic information is required. These instances occur when a **SlipDatum** cell is dumped, either as part of a list dump ((SlipHeader&)X.dumpList()) or when an individual cell is dumped ((SlipDatum&)X.dump(), (SlipReader&)X.dump(), (SlipSequencer&)X.dump()).
- **getParse():** Returns a pointer to the Application Data **parse** method. When the User Data class is registered prior to input, the **getParse()** method is used to store a pointer to the the **parse()** method. Each time an application data definition is found in the input, the **parse()** method is called.
- **getName():** Returns the application supplied data name or the default name (“nullName”). The name field is an invariant property of the base class inherited by all derived classes. If no name is provided, then the default “nullName” is supplied.
- **parse(SlipHeader&):** Converts a list into an internal format needed for a Application Data object. This method is registered prior to input processing (using **getParse()**) and used during input processing to convert input data into a form needed for the Application Data object. The method can be static or dynamic, but if dynamic the object registered for input must be retained until after input is complete. This method should be a factory which given a list returns a SlipDatum Application Data cell. That is, input to the method is passed the SlipHeader of a (possibly empty) list.

  It is the responsibility of the **parse()** method to delete (or use) the input list. If the input list is converted to some internal form and not further used, the **parse()** method must delete it. If the input list is put on another list as a sublist, then the input list must be deleted (see the SlipHeader Section). If a pointer or reference to the input list is used then the input list must not be deleted.
- **remove():** Application specific **delete** synonym. **remove()** deletes the data object. The semantics of delete depend on the decision made for copy, and management of space is the responsibility of the application. Space can not be recovered to the SLIP AVSL unless it is a

SLIP list. The process of deletion, creation, and copy include the name of the Application Data Class. Some hints:

  - Constant: If the data is persistent and **copy()** returns a pointer to the single and only instance of the data, then **remove()** should do nothing or should provide a mechanism to detect the last instance of use and delete the data at that time.
  - Non-Constant: If the data persistency is to be controlled by the longevity of its **SlipDatum** container, then **remove()** must delete the data. The application provided **copy()** method must create a new copy of the the application data on each instance that SLIP requires a copy. This means that each SlipDatum object containing the 'same' SlipPointer derived object actually contains a unique instance of the object.

- **toString():** Pretty-printed string representing the stored data. The representational format shows an application specific meaning. The output is not meant to be input. It only serves to allow the Application Data Type to be attractively rendered.
- **write()**: The format is output by SLIP whenever a list is output (printList()) for the application explicitly invokes this function with (SlipDatum&)X.toString(). The output must be an ASCII formatted list such as '(1 2 (3.0 “apple”) **true**)', where parse() can reassemble the list into appropriate application data.

The SlipPointer ADT is given in Figure 4.4-1: Application Data Type Base Class

```
class SlipPointer {
public:
   SlipPointer(const string* name = &nullName);
   virtual ~SlipPointer();
   virtual SlipPointer*  copy()           = 0;
   virtual string        dump()     const = 0;
   virtual void          remove()   const = 0;
   virtual string        toString() const = 0;
   const   string* const getName()  const;
   virtual string        write()    const = 0;
};
```

Figure 4.4-1: Application Data Type Base Class

4.4.1 Strings

There are two types of *string*s in SLIP Application Data Types recognized, both derived from **SlipPointer** (see Figure 4.4.1-1: SlilpString Inheritance).

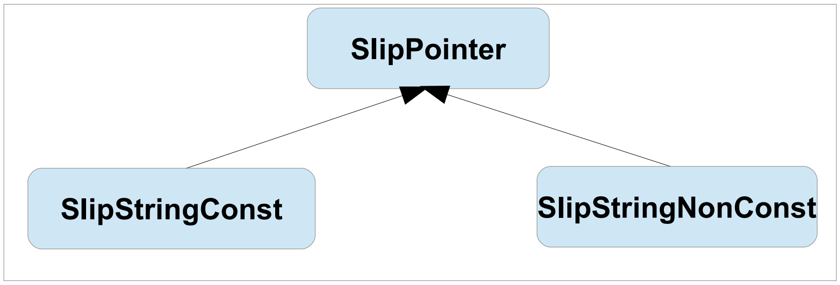

Figure 4.4.1-1: SlilpString Inheritance

SlipStringConst is a predefined data type representing constant *string*s. From the application perspective, the application wants to control the deletion of the indicated *string*, and/or the application wants each **SlipDatum** cell referencing the *string* to reference the same string. Any modification to the string is seen in all **SlipDatum** cells. The default c**opy()** method copies a reference to the same string. The default **remove()** method does nothing.

The **SlipStringNonConst** is a predefined data type representing *string*s that the application wants to relinquish memory control to SLIP. In this case, **copy()** creates a new instance of the *string* and **remove()** deletes the string. The lifetime of the input string is the lifetime of the containing **SlipDatum** cell.

In both cases the reference to 'const' refers to the treatment of the string within SLIP. For **SlipStringConst** there is one instance of the string in all **SlipDatum** copies, and a copy of the string in each **SlipDatum** instance for **SlipStringNonConst** object.

Creation of an instance of a SLIP string requires that the application provides information as to what type of Application Data Type is required, **SlipStringConst** or **SlipStringNonConst**. This can occur in replace, insert, and **SlipDatum** construction. The argument list in all these cases is “const string&, bool constFlag=**false**”, where **false** represents a **SlipStringNonConst** and **true** represents a **SlipStringConst**. The default is to a **SlipStringNonConst**. The list of methods and constructors is given in Table 4.4.1-1 Application Data Type string Creation.

Table 4.4.1-1 Application Data Type *string* Creation

return	**method()**	**Description**
SlipDatum*	SlipDatum(const *string*&, bool constFlag=**false)**	Constructor
SlipDatum*	SlipDatum(const *string**, bool constFlag=**false)**	Constructor

Table 4.4.1-1 Application Data Type *string* Creation

return	**method()**	**Description**
SlipDatum&	insLeft(const *string*&, bool constFlag=**false**)	New SlipDatum returned
SlipDatum&	insLeft(const *string**, bool constFlag=**false**)	New SlipDatum returned
SlipDatum&	insRight(const *string&*, bool constFlag=**false**)	New SlipDatum returned
SlipDatum&	insRight(const *string**, bool constFlag=**false**)	New SlipDatum returned
SlipDatum&	replace(const *string*&, bool constFlag=**false**)	New SlipDatum returned
SlipDatum&	replace(const *string**, bool constFlag=**false**)	New SlipDatum returned

Assignment assumes that the RHS string requires a **SlipStringNonConst** Application Data Type. That is:

SlipDatum X = (string&)Y; and
SlipDatum X = (string*)Y;

create new **SlipDatum** objects with a data type of string and a value of **SlipStringNonConst.** Note that the examples show that a **SlipDatum** object an b e created on the runtime stack.

As a **SlipDatum** object, all the **SlipDatum** operations are available to *string*s, in addition, logical comparison between *string*s are supported.

4.4.2 Application Data Type

The Application Data Type is an extension of the capabilities provided for string types. Application data has the ability to enjoy any application defined data structure and to optionally include the application data in arithmetic, compound assignment, bit, boolean, cast, logical, and unary operations, that is, any operation that the internal data types can participate in.

Application data can be conceptualized as the SlipDatum object being the container for both application operations and application data, as illustrated in Figure 4.4.2-1: Application Framework. The data (SlipPointer class object) is required, the operations (SlipPtrOp class) is not. The SlipPointer class is an ADT and must be the base class for the application data class. The SlipPtrOp class is not an ADT and all operations are defaulted to fail.

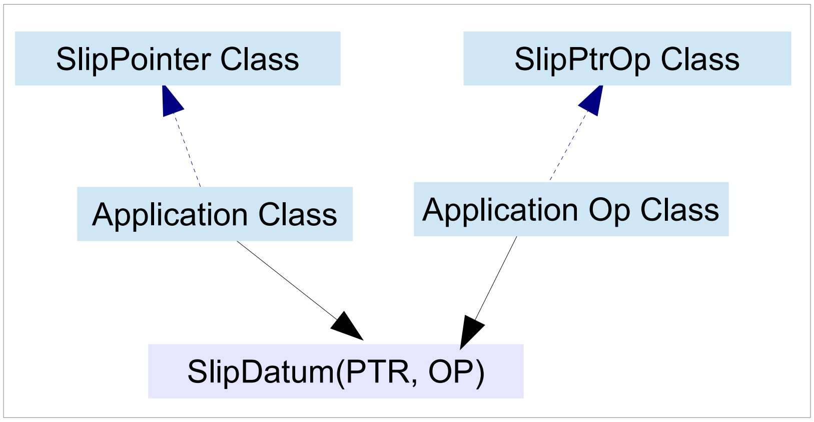

Figure 4.4.2-1: Application Framework

A copy operation is performed in two steps. In the first step SLIP copies the class operations (see Table 4.4.2-1 Application Data Object Creation), in the second step the application provided **copy()** method is called. The application must provide the contract methods given in **SlipPointer** and must understand their impact. **copy()** takes immediate effect at the time it is called. When SLIP requires a copy of the application data, the application provided method is expected to immediately deliver the copy for use, and to be able, at that time, to do any immediate activity required by the application.

At the time that a list or **SlipDatum** object is deleted **remove()** is not called. Removal is deferred until the **SlipDatum** cell containing the application data is needed, and at this time, **remove()** is executed. This may be at some distant time during processing or never. Consider an AVSL with 9,999 cells and a single SlipDatum object with application data. When the SlipDatum object is deleted, the AVSL will contain 10,000 cells. **remove()** will be executed after it's 10,000 predecessor cells have been exhausted. Notifications depending on immediate notification when a cell is placed in the AVSL will not work properly.

Once **remove()** is executed, the application is responsible for recovery of application data. SLIP does not recover this data and the data is never put into the AVSL..

The semantics of **copy()** and **remove()** are application defined. If the rationale used for *string*s applies, the **copy()** and **remove()** work in tandem. If the application requires one copy of the application data then **copy()** will return a reference to this copy and **remove()** will do nothing. If the application allows multiple copies to exist, each one independent of any other, then **copy()** will create a new object with the data, and **remove()** will cause the application to delete the data.

It is important to note that **dump()** provides an internal representation of the SlipDatum cell and then appends the application information. The application data provided should be specific and meaningful to applications, and should be devoid of SLIP-centric data.

The application implementation of **toString()** should return a pretty-printed version of the application data. There's no point in making it look ugly. SLIP has no preconditions of format.

If the application has several different Application Data Types in use, then **getName()** serves to distinguish between them at the application level. There is no requirement that this field be used for this purpose, and if not, the default value of “nullName” is returned. However, if not used, then each data dump will have “nullName” given as the SlipDatum data type for each defaulted Application Data Types and input of a list will all default to the application supplied “nullName” **parse()** method. If one size doesn't fit all, supply a unique string.

SLIP provides the header file, SlipPtrOp.h, as the base class for use in supporting Application Data Type operations. SLIP treats operations as class properties shared by all objects of the same class. For example, if the data type of two SlipDatum cells is double, then the two SlipDatum cells perform the same operations in the same way and refer to the same operator instantiation. It is recommended that each Application Data Type have their own static objects with operations defined. SlipDatum cells with the same data type will reference the same operator properties.

An association between Application Data Type and Application Data Type operations is made during object creation (see Table 4.4.2-1 Application Data Object Creation). The default is to disallow all operations except comparison and assignment. SLIP retains the association after object creation in all SLIP operations.

Table 4.4.2-1 Application Data Object Creation

return	method	Description
SlipDatum*	SlipDatum(const *PTR*, void* operation=null**)**	Constructor
SlipDatum&	insLeft(const *PTR*, void* operation=null)	New SlipDatum returned
SlipDatum&	insRight(const *PTR*, void* operation=null)	New SlipDatum returned
SlipDatum&	replace(const *PTR*, void* operation=null)	New SlipDatum returned

The SLIP framework supports both X op Y and Y op X, where X is a SlipDatum object and Y is a literal, C++ variable, or SlipDatum object. Where the operator, op, is commutative, that is, X op Y will yield the same result as Y op X, SLIP provides an interface compatible with the applications understanding of operations within C++. Where the operations are non-commutative, X op Y and Y op X yield different values, the application needs to provide bodies for special methods within SlipPtrOp.h (see Table 4.4.2-2 Anti-commutative Methods).

As an example:

X + Y and Y + X are commutative and yield the same value.
X – Y and Y – X are not commutative and will yield different values

Table 4.4.2-2 Anti-commutative Methods

result	method	operation
SlipDatum	subOP(const *long*, const SlipDatum&)	(*long*)Y - X
SlipDatum	subUOP(const *unsigned long*, const SlipDatum&)	(*unsigned long*)Y - X
SlipDatum	subDOP(const *double*, const SlipDatum&)	(*double*)Y - X
SlipDatum	divOP(const *long*, const SlipDatum&)	(*long*)Y / X
SlipDatum	divUOP(const *unsigned long*, const SlipDatum&)	(*unsigned long*)Y / X
SlipDatum	divDOP(const *double*, const SlipDatum&)	(*double*)Y / X
SlipDatum	modOP(const *long*, const SlipDatum&)	(*long*)Y % X
SlipDatum	modUOP(const *unsigned long*, const SlipDatum&)	(*unsigned long*)Y % X
long	shlOP(const *long*, const SlipDatum&)	(*long*)Y << X
long	shrOP(const *long*, const SlipDatum&)	(*long*)Y ->> X

5 .0 Iterators

There are two types of iterators in SLIP, a structured iterator and a linear iterator. The structured iterator provides a means to traverse sublists (lists contained with lists) and a memory to enable return to a containing list. The linear iterator provides a means to traverse sublists but no means to return to a containing list.

The iterators provide a consistent view of a list and cells within a list. If a cell is deleted under iterator control, then the iterator uses the preceding cell as the object. If a cell is deleted outside of iterator control, then the iterator contains a stale reference. Under iterator control there is a consistent view of list access. The 'next' cell is always preserved under deletion.

The watchword is that the iterators attempt to provide a consistent view of a list under change.

Iterators are not synchronized. Although each iterator provides a known and consistent interaction with a list under change, this view is supported only for the current iterator and not for multiple iterators in the same list. So although iterators provide quite a bit of support, some native wit is also required.

Iterator functionality is imposed by design. The original SLIP(1)(3) used different method names for a reader and a sequencer, and the sequencer was could not descend into a sublist. These restrictions have been removed. Both reader and sequencer use the same method names to perform the same logical functions, and sequencers can enter sublists as readily as readers (although there doesn't seem to be a good rationale for not using a reader to enter a sublist.)

One other notable difference between the initial SLIP design and the current one is that the current implementation supports consistency and the original SLIP required the application to implement consistency. Operations taken within a list, such as deletion of the current cell, which cause the reader or sequencer to contain a stale pointer are avoided by performing these operations under iterator control.

Common operations are presented below. Of note is that SLIP operations which can be interpreted as either applying to a list or a cell are always interpreted as applying to a list. In the sequencer, there are no list operations supported as part of the iterator. The application is required to know that the iterator references a list in order to do list operations. In a reader unambiguous list operations are supported. Therefore, list centric operations are not common.

Iteration is called advance. Each advance method returns a reference to itself after execution, updated with the results of the advance. In this case, a SlipReader returns a reference to an updated SlipReader and a SlipSequencer returns a reference to an updated SlipSequencer.

There are two types of advance:

Linear (L) search in the current list only.
Structural (S) search in the current list and any sublists.

There are three attributes that can be used in an iteration:

Element (E) advance until a SlipDatum cell is found.

Name (N) advance until a SlipSublist cell is found.
Word (W) advance to the next cell.

There are two directional attributes:

Right (R) advance to the following list cell.
Left (L) advance to the preceding list cell.

Advance on a word is an operation which always terminates after a single step is executed. Advance on an element or a name is a search operation which terminates when either the given item is found or when a list header (SlipHeader object) is found. If the search is linear then on failure a search will terminate at the list header (SlipHeader object) of the current list. If the search is structural then on failure the search will terminate at the list header of the topmost list for a reader and at the bottommost list for a sequencer.

On structural advances a depth first search is performed. Each time that a sublist (name) is traversed, the subordinate list is entered. For a sequencer the subordinate list never returns to its parent list. For a reader, a return to the parent list is made whenever a search or advance continues through the list header. Note that an word advance is not a search. It will always stop at a list header before the next advance.

The advance functions are given in Table 5-1.

Table 5-1 Advance Methods

method()	**Description**
advanceLER()	Advance Linear Element Right. Search for a SlipDatum cell in the current list
advanceLNR()	Advance Linear Name Right. Search for a SlipSublist cell in the current list.
advanceLWR()	Advance Linear Word Right. Single step to the next cell.
advanceLEL()	Advance Linear Element Left. Search for a SlipDatum cell in the current list
advanceLNL()	Advance Linear Name Left. Search for a SlipSublist cell in the current list.
advanceLWL()	Advance Linear Word Left. Single step to the next cell.
advanceSER()	Advance Structural Element Right. Search for a SlipDatum cell in any list.
advanceSNR()	Advance Structural Name Right. Search for a SlipSublist cell in any list.
advanceSWR()	Advance Structural Word Right. Single step to the next cell.
advanceSEL()	Advance Structural Element Left. Search for a SlipDatum cell in the any list.

Table 5-1 Advance Methods

method()	Description
advanceSNL()	Advance Structural Name Left. Search for a SlipSublist cell in any list.
advanceSWL()	Advance Structural Word Left. Single step to the next cell.

Cell based methods are supported by both iterators. These include the insert (**insRight()** and **insLeft()**) and the replace methods, assignment and logical operations. The miscellaneous **toString()** and **dump()** methods output information on the current SlipCell and not the iterators. A list of all common methods are given in Table 2.3-1 Assignment Operations, Table 3.2-1 Replace Methods and Table 3.3-1 Insert Methods. Table 3.4-1 Move Methods applies to **moveLeft()** and **moveRight()**, and Table 3.6-1 Miscellaneous Methods applies to **dump()**, **getClassType()** and **toString()**. The format for execution of methods is X.method(), assignments are given by X = value, and relational operations by X == value, where X is either a SlipReader or SlipSequencer.

The return types for the incorporated methods from other sections is the same as defined in the referenced sections. For **moveListRight()** and **moveListLeft()**, the new methods presented in Table 5-2 Miscellaneous Iterator Methods have an iterator as a return type.

The interrogatories supported in Table 3.1-1 Interrogatories along with the property **getName()** described in Table 3.6-1 Miscellaneous Methods. Interrogatories guaranteed to be true (**isAVSL()**) or guaranteed to be untrue (**isDeleted()**, **isTemp()**, **isUnlinked()**) are not defined.

In addition, each iterator supports all the inserts, moves, move lists, and replaces given in Table 3.2-1 Replace Methods, Table 3.3-1 Insert Methods, Table 3.4-1 Move Methods, Table 4.1-1 SlipHeader Constructors and Destructors, Table 4.4.1-1 Application Data Type string Creation, Table 4.4.2-1 Application Data Object Creation. The format and returns are the same as defined. Invocation using the current iterator, such as (SlipReader&)X.method() or (SlipSequencer&)X.method(). The current cell of the iterator is returned and is unchanged.

The remaining common and unique methods are given in Table 5-2. Either iterator, **I** = {SlipReader, SlipSequencer}, can be used in **I.method()** or returned as I&.

Table 5-2 Miscellaneous Iterator Methods

return	I.method()	Description
SlipCell&	currentCell()	Return a reference to the list cell currently accessed

Table 5-2 Miscellaneous Iterator Methods

return	I.method()	Description
		by the iterator. The current cell can be a header, sublist reference, or datum cell.
I&	deleteCell()	Delete the current cell and fixup the sequencer. After the delete operation, the sequencer will reference the list cell preceeding the deleted cell. This operation is equivalent to performing an advanceLWL() and then deleting the next cell.
SlipCell&	insLeft(SlipReader&)	A copy of the current cell referenced in the reader is inserted. Return a reference to the inserted cell.
SlipCell&	insLeft(SlipSequencer&)	A copy of the current cell referenced in the sequencer is inserted. Return a reference to the inserted cell.
SlipCell&	insRight(SlipReader&)	A copy of the current cell referenced in the reader is inserted. Return a reference to the inserted cell.
SlipCell&	insRight(SlipSequencer&)	A copy of the current cell referenced in the sequencer is inserted. Return a reference to the inserted cell.
SlipCell&	replace(SlipReader&)	The current cell referenced in the reader replaces the current cell referenced in **I**. Return a reference to the new cell.
SlipCell&	moveLeft(SlipReader&)	Move the reader current cell to the left of the iterator current cell. Return the current cell. • If the input current cell is a list then the list body is moved to the left of the iterator current cell and the input iterator references an empty list. • Otherwise the input current cell is moved and the input iterator references its predecessor cell.
SlipCell&	moveLeft(SlipSequencer&)	Move the reader current cell to the left of the iterator current cell. Return the current cell. • If the input current cell is a list then the list body is moved to the left of the iterator current cell and the input iterator references

Table 5-2 Miscellaneous Iterator Methods

return	I.method()	Description
		an empty list. • Otherwise the input current cell is moved and the input iterator references its predecessor cell.
I&	moveListLeft(SlipReader&)	Move the reader current list to the left of the iterator current cell. Return the current cell. • The list body is moved to the left of the iterator current cell and the reader current list and current cell reference an empty list.
I&	moveListLeft(SlipSequencer&)	Move the sequencer list to the left of the iterator current cell. Return the current cell. • If the input current cell is a list list body is moved to the left of the iterator current cell and the reader current list and current cell reference an empty list. • Otherwise it is an error,
I&	moveListRight(SlipReader&)	Move the reader current list to the right of the iterator current cell. Return the current cell. • The list body is moved to the left of the iterator current cell and the reader current list and current cell reference an empty list.
I&	moveListRight(SlipSequencer&)	Move the sequencer list to the right of the iterator current cell. Return the current cell. • If the input current cell is a list list body is moved to the left of the iterator current cell and the reader current list and current cell reference an empty list. • Otherwise it is an error,
SlipCell&	moveRight(SlipReader&)	Move the reader current cell to the right of the iterator current cell. Return the current cell. • If the input current cell is a list then the list body is moved to the right of the iterator current cell and the input iterator references an empty list. • Otherwise the input current cell is moved

Table 5-2 Miscellaneous Iterator Methods

return	I.method()	Description
		and the input iterator references its predecessor cell.
SlipCell&	moveRight(SlipSequencer&)	Move the reader current cell to the right of the iterator current cell. Return the current cell. • If the input current cell is a list then the list body is moved to the right of the iterator current cell and the input iterator references an empty list. • Otherwise the input current cell is moved and the input iterator references its predecessor cell.
SlipCell&	replace(SlipSequencer&)	The current cell referenced in the sequencer replaces the current cell referenced in **I**. Return a reference to the new cell.
SlipCell&	unlink()	The current cell is unlinked from the list and the iterator performs an advanceLWL() to reference its preceding cell. Return a reference to the unlinked cell.

An example of usage is given in Example 5-1.

```
SlipHeader H1 = new SlipHeader();
SlipHeader H2 = new SlipHeader();
   H1.enqueue((long)0);     // H1 = (0)
   H1.enqueue((long)2);     // H1 = (0 2)
   H1.enqueue(H2);          // H1 = (0 2 () )
   H1.enqueue((long)4);     // H1 = (0 2 () 4 )
   H1.enqueue((long)6);     // H1 = (0 2 () 4 6)

SlipSequencer S1 = new SlipSequencer(H1);
SlipReader    R1 = new SlipReader(R1);
   R1.advanceLER();        // data = 0
   S1.advanceLER();        // data = 0
   S1.advanceLER();        // data = 2

   do {        // R1 += S1
      R1.currentCell() += S1.currentCell();
      cout << R1.toString() << “ += “ << S1.toString() <<
endl;
      S1 = R1.currentCell() + S1.currentCell()
      S1.advanceLER();
   } while(!S1.isHeader())

if (R1.isNumber()) cout << “R1 is a number ” R1.getName()
                        << endl;
if (S1.isHeader()) cout << “S1 is a header” << endl;
while(!(S1.advanceLER()).isHeader())
   cout << S1.currentCell().toString() <<  ' ';
cout << endl;
```

Example 5-1 Iterator Example

The output from Example 5.1 is:

```
2 += 2
6 += 4
12 += 6
R1 is a number LONG
S1 is a header
4 10 18
```

An extended example of the iterators for **`list = (a b (c (d) e ) f)`** is given.

The notation used is D(value) represents a **SlipDatum** value, H() represents a **SlipHeader** defined list, and N(H()) represents a **SlipSublist** reference, N() to the list H(). We transform the above list to the following linear relation:

`list = L1( D(a) D(b) N(L2(D(c) N(L3(D(d))) D(e)) D(f))`

The spatial relation representing this list is:

```
L1( D(a) D(b) N() D(f) )
              |
              |
              o L2( D(c) N() D(e) )
                         |
                         |
                         o L3( D(d) )
```

The following examples show the effect of advances for each iterator over the list. For linear advances the cell referenced is the same for the reader and sequencer. For structural advances they are very different.

In our example there are two sublists, N(L2) and N(L3) representing the sublist cells referencing lists L2 and L3 respectively. Lists are shown as L1, L2 or L3, and data cells are shown as D().

In the examples, the reader maintains a reference to the list, **List**, and its current cell, **Cell**, and the nesting depth, **Depth**. The sequencer contains a reference to its current cell, **Cell**.

All iterators start at L1.

Method	**Reader**			**Sequencer**
advanceLER	**List**	**Cell**	**Depth**	**Cell**
advanceLER	L1	D(a)	0	D(a)
advanceLER	L1	D(b)	0	D(b)
advanceLER	L1	D(f)	0	D(f)
advanceLER	L1	L1	0	L1

Example 5-2 advanceLER

Method	**Reader**			**Sequencer**
advanceLEL	**List**	**Cell**	**Depth**	**Cell**
advanceLEL	L1	D(f)	0	D(f)
advanceLEL	L1	D(b)	0	D(b)
advanceLEL	L1	D(a)	0	D(a)
advanceLEL	L1	L1	0	L1

Example 5-3 advanceLEL

Method	**Reader**			**Sequencer**
advanceLNR	**List**	**Cell**	**Depth**	**Cell**
advanceLNR	L1	N(L2)	0	N(L2)
advanceLNR	L1	L1	0	L1

Example 5-4 advanceLNR

Method	**Reader**			**Sequencer**
advanceLNL	**List**	**Cell**	**Depth**	**Cell**
advanceLNL	L1	N(L2)	0	N(L2)
advanceLNL	L1	L1	0	L1

Example 5-5 advanceLNL

Method	**Reader**			**Sequencer**
advanceLWR	**List**	**Cell**	**Depth**	**Cell**
advanceLWR	L1	D(a)	0	D(a)
advanceLWR	L1	D(b)	0	D(b)
advanceLWR	L1	N(L2)	0	N(L2)
advanceLWR	L1	D(f)	0	D(f)
advanceLWR	L1	L1	0	L1

Example 5-6 advanceLW*R*

Method	**Reader**			**Sequencer**
advanceLWL	**List**	**Cell**	**Depth**	**Cell**
advanceLWL	L1	D(f)	0	D(f)
advanceLWL	L1	N(L2)	0	N(L2)
advanceLWL	L1	D(b)	0	D(b)
advanceLWL	L1	D(a)	0	D(a)
advanceLWL	L1	L1	0	L1

Example 5-7 advanceLWL

Method	**Reader**			**Sequencer**
advanceSER	**List**	**Cell**	**Depth**	**Cell**
advanceSER	L1	D(a)	0	D(a)
advanceSER	L1	D(b)	0	D(b)
advanceSER	L2	D(c)	1	D(c)
advanceSER	L3	D(d)	2	D(d)
advanceSER	L2	D(e)	1	L3
advanceSER	L1	D(f)	0	L3
advanceSER	L1	L1	0	L3

Example 5-8 advanceSER

Method	Reader			Sequencer
advanceSEL	**List**	**Cell**	**Depth**	**Cell**
advanceSEL	L1	D(f)	0	D(f)
advanceSEL	L2	D(e)	1	D(e)
advanceSEL	L3	D(d)	2	D(d)
advanceSEL	L2	D(c)	1	L3
advanceSEL	L1	D(b)	0	D(d)
advanceSEL	L1	D(a)	0	L3
advanceSEL	L1	L1	0	D(d)

Example 5-9 advanceSEL

Method	Reader			Sequencer
advanceSNR	**List**	**Cell**	**Depth**	**Cell**
advanceSNR	L1	N(L2)	0	N(L2)
advanceSNR	L2	N(L3)	1	N(L3)
advanceSNR	L1	L1)	0	L3

Example 5-10 advanceSNR

Method	Reader			Sequencer
advanceSNL	**List**	**Cell**	**Depth**	**Cell**
advanceSNL	L1	N(L2)	0	N(L2)
advanceSNL	L2	N(L3)	1	N(L3)
advanceSNL	L1	L1)	0	L3

Example 5-11 advanceSNL

Method	**Reader**			**Sequencer**
advanceSWR	**List**	**Cell**	**Depth**	**Cell**
advanceSWR	L1	D(a)	0	D(a)
advanceSWR	L1	D(b)	0	D(b)
advanceSWR	L1	N(L2)	0	N(L2)
advanceSWR	L2	D(c)	1	D(c)
advanceSWR	L2	N(L3)	1	N(L3)
advanceSWR	L3	D(d)	2	D(d)
advanceSWR	L2	D(e)	1	L3
advanceSWR	L1	D(f)	0	D(d)
advanceSWR	L1	L1	0	L3

Example 5-12 advanceSWR

Method	**Reader**			**Sequencer**
advanceSWL	**List**	**Cell**	**Depth**	**Cell**
advanceSWL	L1	D(f)	0	D(f)
advanceSWL	L1	N(L2)	0	N(L2)
advanceSWL	L2	D(e)	1	D(e)
advanceSWL	L2	N(L3)	1	N(L3)
advanceSWL	L3	D(d)	2	D(d)
advanceSWL	L2	D(c)	1	L3
advanceSWL	L1	D(b)	0	D(d)
advanceSWL	L1	D(a)	0	L3
advanceSWL	L1	L1	0	D(d)

Example 5-13 advanceSWL

5.1 SlipSequencer

This is a fast iterator designed primarily for quick iteration of a list, without iteration over subordinate lists. It's functionality has been extended to include all of the advance methods used in the reader, which includes entry into subordinate lists. However, once the sequencer enters a subordinate list, it can not return to the lists' parent.

All of the methods described in Section 5.0 are available to the SlipSequencer. The SlipSequencer constructors and destructors are defined in Table 5.1-1 and the unique methods are defined in Table 5.1-2.

Table 5.1-1 Constructors and Destructors

return	method()	Description
SlipSequencer*	SlipSequencer(SlipHeader&)	Sequencer current cell references SlipHeader.
SlipSequencer*	SlipSequencer(SlipSublist&)	Sequencer current cell references SlipSublist header reference.
void	delete	Deletes the sequencer. The current cell is unaffected.

Table 5.1-2 Unique Selector Methods

result	method()	Description
SlipSequencer&	reset(SlipCell&)	Reset the current cell to the SlipCell. The SlipCell must be part of a list.
SlipSequencer&	reset(SlipHeader&)	Reset the current cell to the SlipHeader.
SlipSequencer&	reset(SlipReader&)	Reset the current cell to the SlipReader current cell. This generates two iterators for the same list.
SlipSequencer&	reset(SlipSequencer&)	Reset the current cell to the SlipSequencer current cell. This generates two iterators for the same list.

5.2 SlipReader

The **SlipReader** is a heavy duty iterator. It allows entry into a subordinate list and return from a subordinate list to its parent. The performance for linear advances are about the same as for the **SlipSequencer**. The performance for structural advances are typically greater than that for the sequencer.

Each SlipReader contains the following application accessible properties:

- A reference to the current list.

  - As each list is entered, it becomes the current list.
  - When a subordinate list is entered during a structural advance, the previous SlipReader is stacked and a new SlipReader cell is created with the new list being the current list and current cell, and the depth incremented by 1'
- A reference to the current cell. Advances cause the current cell reference to change. On entry to a subordinate list, the current cell will reference the list header. On exit from a subordinate list, the current cell will be reference the sublist cell referencing the subordinate list advanced according the advancing method.
- The depth of the current list, where the depth of the topmost list is 0.

Structural advance examples in Section 5.0 Iterators illustrates the operations when entering and leaving a subordinate list.

The SlipReader iterator supports all the methods given in Section 5.0 and in the SlipHeader, Sections 4.1, 4.1.1 and 4.1.2. These methods are defined in Table 4.1-1 SlipHeader Constructors and Destructors, Error: Reference source not found, Error: Reference source not found, Error: Reference source not found, and Table 4.1.2-1 Descriptor List Methods.

The constructor and destructor methods are given in Error: Reference source not found. The SlipReader may be created from the stack or heap (with **new**). If it is created from the heap, it must be deleted.

Table 5.2-1 SlipReader Constructor/Destructors

return	method()	Description
SlipReader&	SlipReader(SlipHeader&)	Creates a SlipReader with the current list = current header = SlipHeader and the depth = 0.
SlipoReader&	SlipReader(SlipSublist&)	Creates a SlipReader with the current list = current header = SlipSublist.getHeader() and the depth = 0.
void	delete	Deletes all the good that we have done.

The methods unique to the SlipReader are given in Table 6.2-2.

Table 5.2-2 Unique SlipReader Methods

return	(SlipReader&) X.method()	Description
SlipHeader&	currentList()	Return a reference to the current list header.
short	listDepth()	Return the list depth. A depth of zero is the topmost list.
SlipHeader&	reset()	Reset the current cell to the current list. Note that this is different than the SlipSequencer resets.
SlipReader&	resetTop()	Return to parent list. The current list will be a reference to the topmost list. The current cell will be to the sublist cell entered the subordinate list chain. The depth will be 0. In other words, return to where you entered the list chain. This is equivalent the performing upLevel() listDepth() times.
SlipReader&	upLevel()	The current list will be a reference to the parent list. The current cell will be to the sublist cell which referenced the subordinate list. The depth will be the depth of the parent list. If the current list is the topmost list, do nothing.

6 .0 Errors

6.1 Theory of Operation

Errors are only detecting during SLIP operations. It is assumed that an error causes the application to enter an unreliable state, and although the effect of the error may not be felt at the time of discovery, continued operation will lead to erroneous results or a catastrophic failure. The default action on error detection is to throw a **slipException** and enable the application to degrade gracefully.

The default action can be controlled by the application. Each error message's internal state can be altered from issuing an exception (slip::eException) to issuing a warning but not an exception (slip::eWarning) or to ignoring the error entirely (slip::eIgnore). SLIP attempts to perform internal actions to allow the application to continue and, as appropriate, return a software fault indication to the user. Where this can not be done, SLIP will abort the application. Changing the overall behavior does not mitigate the potential for catastrophic effects. The runtime detected error is still an error, and whatever consequences occur from this error are still present.

The application can assume control over all error detection by providing an application callback function. On detection of any error, the application callback function is invoked and the generated error message and the error name are provided to the function. The application can then output the diagnostic message and perform application specific error recovery. On return from the callback function the application can flag the calling error function to either throw an exception or to ignore the error. The SLIP diagnostic function will not output a message.

There are no limitation on the number of times a message state is changed, or on how many messages are changed. There are no limitations on the number of times the application callback function is set, but only the last callback function will be active.

6.2 Error Message Format

The output message format is:

```
Filename: LineNumber: in Function ErrorNumber  MessageText
        [CellDump]
        [CellDump]
```

Where the fields are defined as:

Filename	File name where error was detected or reported
LineNumber	Line number of error post
Function	Function posting the error in the file
ErrorNumber	Error number. Error numbers are formatted as 'E' followed by a 4 digit number, as in “E2112”.
MessageText	The message text describing the error.
[CellDump]	Optional SLIP cell dumps. Format depends on cell type.

6.3 Application Callback function

The application can intercept all diagnostic messages through an application provided callback function. The callback function gets control each time SLIP attempts to output a diagnostic message. The input arguments allow identification of the message being output, the filename and line number of the message initiator, and the formatted diagnostic message.

The application can elect to perform application specific error recovery and message output before returning. On return, the application specifies that the SLIP software throws or does not throw a slipException.

The prototype callback function is defined as:

```
bool errorCallback((string filename, int lineno, SlipErr::Error err, string message)
```

where:

filename	Filename where error detected or message posted.
lineno	Line number where error detected or message posted.
err	Message object.
message	Formatted message.
return	**true** causes a slipException to be issued. **false** ignores the error. In neither case is the formatted message output.

6.4 Message Object Format (SlipErr)

The message object is passed to the callback function and message content is available through public methods. The methods provide access to the unformatted message, the message state, the index of the message body in the array of message bodies, and the message number.

There is a provided method to alter the message state but no means to change the message body, number, or table index, see Table 6.4-1: Message Object Table.

Table 6.4-1: Message Object Table

<table>
<tr><th>return</th><th>SlipErr::Error.funct()</th><th>Description</th></tr>
<tr><td>string</td><td>getErrorNumber()</td><td>The format is 'E' followed by 4 decimal numbers</td></tr>
<tr><td>int</td><td>getIndex()</td><td>Return the message object index into the object table.</td></tr>
<tr><td>string</td><td>getMessage()</td><td>Return the unformatted message string.</td></tr>
<tr><td>errorType</td><td>getState()</td><td>Return the errorType.
<table>
<tr><th colspan="2">slip::errorType</th></tr>
<tr><td>eIgnore</td><td>Ignore message</td></tr>
<tr><td>eWarning</td><td>Output a formatted message</td></tr>
<tr><td>eException</td><td>Output a formatted message & throw a slipException</td></tr>
</table>
</td></tr>
<tr><td>bool</td><td>isException()</td><td>true if errorType == eException</td></tr>
<tr><td>bool</td><td>isIgnore()</td><td>true if errorType == eIgnore</td></tr>
<tr><td>bool</td><td>isWarning()</td><td>true if errorType == eWarning</td></tr>
<tr><td>void</td><td>setState(errorType)</td><td>Set the message error state</td></tr>
</table>

6.5 SlipException

The SlipException object provides information useful in determining the cause of the exception and methods required to determine what action to take. If the application callback function throws a SlipException it must populate the object. Table 6.5-1: SlipException Constructor shows the constructor parameters.

Table 6.5-1: SlipException Constructor

type	parameters	description
SlipErr::Error	base	Message property object, see Table 6.4-1: Message Object Table
string	message	Diagnostic message
SlipCell*	cell1	Pointer to a cell or **null**
SlipCell*	cell2	Pointer to a cell or **null**

The base is used by the SlipErr software to construct a formatted message for output. If the formatted message contains a SlipCell or SlipCells, then a pointer to these cells are passed to the SlipException object. If there is one SlipCell then it will always be cell1. SlipCells that are missing have **null** as the pointer value.

The methods required to access the exception values are given in Table 6.5-2: SlipException Methods.

Table 6.5-2: SlipException Methods

const	method
SlipErr::Error	getBase()
string&	getMessage()
SlipCell&	getCell1()
SlipCell&	getCell2()
char*	what()

what() returns the same message as **getMessage()**.

7 .0 Slip.h

Slip.h is the application interface to the SLIP API. The header file contains three elements:

1. Inclusion of all required classes. The required and accessible classes for application access to SLIP are included.

2. Overloaded operators with SLIP as the second operand. An example is:
   `bool operator<=(LONG a, SlipCell b);`
3. Utility functions. Some useful (static) methods are provided.

The entire SLIP API is provided.

Table 7-1: slip.h Utility Functions

return	function	description
void	avslHistory(bool)	true turns history on
SlipState	getSlipState()	Return internal SLIP state
void	printAVSL(caption)	Output the AVSL
void	printClassSizes()	Bytes in each SLIP class
void	printFragmentList(caption)	Format the fragment list
void	printMemory(caption)	All AVSL is output
void	printState(string)	AVSL state
errorType	setErrorState(errorType, string)	Change message state property ()
void	sysInfo(ostream&)	SLIP system infrmation
void	slipInit()	Initialize SLIP using default values
void	slipInit(alloc, delta)	Initialize SLIP
errorCallback	setCallBack(errorCallback)	Set application message trap

8 .0 List Input / Output

Input and output functions guarantee that if SLIP writes (W) a list and then reads a list (R) then, W= W(R) and that R = R(W), that is, if the list is written using SLIP, then when it is read the same topology and content will be retrieved, and if the list is read then written then the output file will represent the same list (formatting and list names may be different). Therefore, the ASCII formatted output is a suitable framework for storing and retrieval of information.

The application is given access to constructs which allow a list to be created offline and then input. The guarantees given above apply to well-formed constructs, that is, files which 'follow the rules'.

This section defines the rules for construction of list files and define the format of list files created by SLIP.

8.1 Lexical Elements

8.1.1 Character Set

The allowable character set (outside of quoted tokens) is:

1. `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
   `a b c d e f g h I j k l m n o p q r s t u v w x y z`
2. The ten digits:
   1 2 3 4 5 6 7 8 9 0
3. The special graphics characters:
   { } ( ) # “ ' < > / * . ; $ _
4. The **SPACE**.
5. The horizontal tab (HT).
6. The newline character (NL or \n).

Inside a quoted token the following additional characters are allowed:

1. The remaining graphic characters:
   , / ? ' : | [ ] ` ~ ! @ & _- = +
2. Escaped characters
   \b bell
   \f form feed
   \n new line
   \r carriage return
   \t vertical tab
   \c any escaped character eg. \”, or \'

8.1.2 Whitespace

Where one whitespace character is legal many can be used. The whitespace characters are the set {\a \b \f \n \r \t **SPACE**}. All characters are interpreted as a **SPACE**, and multiple characters are treated as a single **SPACE**.

8.1.3 Comments

Anyplace whitespace is legal, a comment can be used. Comments are treated as a single whitespace character when seen. There are two types of comments:

1. Multiline: A comment bracketed by '/*' and '*/ 'can be wholly contained in one line or can extend across multiple lines.

2. Single line: A comment beginning with '//' extends from its current location until the end of line.

All characters seen from the comment start to its termination are ignored. In the case of multiline comments, this is from the leading '/*' to the trailing '*/'. In the case of single line comments, this is from the comment start '//' to the end of line. Nested comments are not supported. Comments are not saved in a list, they are ignored.

```
( ( /* this is a comment */ ) )

( ( /* this is a
       multiline comment */ ) )

( ( // this is a single line comment
) )
```

Example 8.1.3-1 Comment Example

8.1.4 Tokens

The characters in the alphabet are collected into a token by the lexer. The tokens are the maximum set of characters that can be combined to form a token according to the token rules. There are 4 types of tokens, separators, keywords, identifiers, constants.

8.1.4.1 Separators

The following characters are separators and form a single token:
( ) { } < > #

The legal context for each token is given below. Each separator, except '#', must be balanced. That is, where the left separator appears, the right separator must also appear using the following algorithm:

1. Push the left separator onto a stack.
2. When the right separator is seen, the left separator must be on the stack top. Pop the left separator.

The separators have the following meanings:

1. ( ): Brackets a list. All items between these brackets are part of the same list. Parentheses can be nested, as in “((( ) ( ))”.

2. < >: Brackets a Descriptor list. All items between these brackets are part of the same list. Angle brackets can be nested, as in “< < < > > >”. Unlike lists, brackets must be nested, and can not be disjoint, as in “< < > < > >”.

3. { }: This identifies a named list, as in “{name}”. The brackets can not be nested.

4. #: This identifies a preprocessing statement. The action performed is immediate and the results of the action are processed by the parser.

8.1.4.2 Identifiers

An identifier consists of a leading alphabetic character or an underscore ('_') or dollar sign ('$') followed by zero or more alphanumeric characters or underscores ('_') or dollar signs ('$'). Embedded whitespace is not allowed and there are no limitations on size.

As a regular expression this can be written as:

```
([a-z]|[A-Z]|$|_)([a-z]|[A-Z]|[0-9]|$|_)*
```

```
Legal Names:
_                     // single character name
$                     // single character name
name                  // alphabetic character name
name_                 // alphabetic & special symbols
__na$m123e$_          // alphanumeric & special symbols
x01234                // alphanumeric
name_name             // alphabetic & special symbols

Illegal Names:
na me                 // embedded blank
2name                 // leading numeric character
_name[]               // illegal characters
name-name             // illegal character
```

Example 8.1.4-1: Legal / Illegal Names

8.1.4.3 Keywords

There are two keywords in SLIP, “user”, and “include”. They must both be in lower case and spelled exactly as shown.

8.1.4.4 Constants

SLIP supports several types of constants. There is a one-to-one correspondence with the constants in input/output operations and the typedef's used in runtime applications. The constants and the typedef's are architecturally independent and are designed to have the same content and size from compiler to compiler, and computer to computer. The format corresponds to the general format of C/C++ constants with the constraint that they are SLIP, and not, C/C++ values.

Boolean values are represented as 'true' or 'false' on input or output. When the input list file has 'true' or 'false' a boolean object is created as in SlipDatum((bool)true). During output, a boolean object is output as the literal 'true' or 'false'.

Embedded blanks are not allowed in integral or real number constants.

The integral types are octal, decimal, and hexadecimal.

Octal constants are represented as '0' followed by zero or more digits in the range 0 - 7. For example, 012345 is an octal number but 0123458 is not – 8 is not an octal number.

Integer constants are represented as '1' followed by zero or more digits in the range 0 – 9. For example 12345 is an integer and is different from 012345, an octal number.

Hexadecimal constants are represented as “0x” or “0X”, where '0' is the number zero, followed by one or digits in the range 0 – 9, a – f, A – F. For example 0xDeadBeef is a hexadecimal constant.

Each integral type can be optionally followed by a unsigned specification, 'U', and a size specification, 'C' or 'L', Capitalization is ignored. If 'U' is not specified, then the number is signed. The specifications have the following meaning.

- Signed means that the integral number is stored into a signed SlipDatum character or integer object.
- Unsigned means that the integral number is stored into a nunsigned SlipDatum character or integer object.
- 'C' means that the integral number is stored into a SlipDatum((CHAR) object. Hexadecimal number can not have a 'C' size specification.
- 'L' means that the integral number is stored into a SlipDatum((LONG) object.

The default for a missing unsigned specification is signed, and the default for a missing size specification is 'L'.

Example 8.1.4-2 Integral Constructions, shows the SlipDatum object created for each integral type in the input list file. Output is always an integer, not octal or hexadecimal, so the automatically generated output for each integral type is the number at the left. If the integral type is a signed integer, SlipDatum((LONG)), then the integer output does not include a size or sign specification but defers to the default.

```
Example                Constructed SlipDatum object
012345       ->        SlipDatum((LONG)012345)          // octal number
0123C        →>        SlipDatum((CHAR)0123)            // octal number
012345L      ->        SlipDatum((LONG)012345)          // octal number
012345U      ->        SlipDatum((ULONG)012345)         // octal number
0123UC       ->        SlipDatum((UCHAR)0123)           // octal number
246789       ->        SlipDatum((LONG)246789)          // decimal number
123C         ->        SlipDatum((CHAR)123)             // decimal number
246789L      ->        SlipDatum((LONG)246789)          // decimal number
246789U      ->        SlipDatum((ULONG)246789)         // decimal number
0xAB         ->        SlipDatum((LONG)0xAB)            // hexadecimal number
0xABL        ->        SlipDatum((LONG)0xAB)            // hexadecimal number
0xABUC       ->        SlipDatum((UCHAR)0xAB)           // hexadecimal number
0xABU        ->        SlipDatum((ULONG)0xAB)           // hexadecimal number
```

Example 8.1.4-2 Integral Constructions

SLIP only supports real doubles. The input format for a double is any of “

- #.#, #., .# optional decimal number with a fraction. The decimal point is required.
- #E#, #E+#, #E-#: real number with an exponent. The exponent, 'E', can be upper case or lower case.
- #.E#, ..#E# any combination of the previous two forms.
- Any of the above forms with a sign (+, -).

In a manner similar to integral numbers, real numbers are defined in Example 8.1.4-3 Real Number Constructions. Input from the list input file creates an object of type SlipDatum((LONG)).

```
Example             Constructed SlipDatum object
1.             ->   SlipDatum((DOUBLE)1,)           // floating point number
+1.            ->   SlipDatum((DOUBLE)+1.)          // floating point number
.1             ->   SlipDatum((DOUBLE).1)           // floating point number
-.1            ->   SlipDatum((DOUBLE)-.1)          // floating point number
1.1            ->   SlipDatum((DOUBLE)1.1)          // floating point number
-1.1           ->   SlipDatum((DOUBLE)-1.1)         // floating point number
1E10           ->   SlipDatum((DOUBLE)1E10)         // floating point number
+1E-10         ->   SlipDatum((DOUBLE)+1E-10)       // floating point number
-.1E+10        ->   SlipDatum((DOUBLE)-.1E+10)      // floating point number
```

Example 8.1.4-3 Real Number Constructions

Single characters are input or output as 'c', where 'c' is a character. Strings are multiple characters delimited by “”, as in “multiple characters”. Characters and strings can have embedded blanks.

Character objects are created using SlipDatum((CHAR)'c') and string objects are created using SlipDatum((STRING)”string”) which is equivalent to SlipDatum((STRING)”string”, false). The application loses its ability to share and control strings on input from an input list file. SLIP deletes strings during object deletion, and creates a new instance of a string when a copy operation is warranted.

Strings and characters can have octal, decimal, hexadecimal, and special characters inserted into them. Integral data types do not have sign or size specification. The characters are converted to an 8-bit octet. Strings are output without conversion. Characters are output as an integer for values outside the printable character range. On output format of these numbers and characters is:

- \0#   octal number. Where $0 \le \# \le 7$.

The input list file legal formats are:

- \0# octal number. Where $0 \le \# \le 7$.
- \1# decimal number. Where $0 \le \# \le 9$.
- \x# hexadecimal number. Where $0 \le \# \le 9$ or $a \le \# \le f$ or $A \le \# \le F$..
- \b bell.
- \f form-feed.
- \n newline.
- \t tab.

```
SlipDatum((CHAR)1)->write() => '\1'

'\1'  => SlipDatum((CHAR)1);

'\01' => SlipDatum((CHAR)1);

'\x1' => SlipDatum((CHAR)1);

SlipDatum((STRING)”12\n34”)->write() => 12
                                        34

“12\n34” => SlipDatum((STRING)”12\n34”) // binary '10' replaces \n
```

Example 8.1.4-4 Character/String Input/Output

For convenience types are provided to the application. The meaning of the types is SLIP-centric.

Table 8.1.4-1 Integer Constant Equivalences

INPUT	Application	OUTPUT	cstdint	range
TRUE	`SlipDatum((bool)true)`	TRUE	bool	$0 \le x \le 1$
FALSE	`SlipDatum((bool)false)`	FALSE	bool	$0 \le x \le 1$
'c'	`SlipDatum((CHAR)'c')`	'c'	int8_t	$-128 \le x \le 127$
'\05'C	`SlipDatum((CHAR)'\05')`	5C	int8_t	$-128 \le x \le 127$
'\x41'C	`SlipDatum((CHAR)'\x41')`	A	int8_t	$-128 \le x \le 127$
'\n'C	`SlipDatum((CHAR)'\n')`	10C	int8_t	$-128 \le x \le 127$
'\xFF'C	`SlipDatum((UCHAR)'\xFF')`	-1C	int8_t	$-128 \le x \le 127$
'c'	`SlipDatum((UCHAR)'c')`	'c'	uint8_t	$0 \le x \le 255$

Table 8.1.4-1 Integer Constant Equivalences

'\05'UC	`SlipDatum((UCHAR)'\05')`	5UC	uint8_t	$0 \le x \le 255$
'\x41'UC	`SlipDatum((UCHAR)'\x41')`	A	uint8_t	$0 \le x \le 255$
'\n'UC	`SlipDatum((UCHAR)'\n')`	10UC	uint8_t	$0 \le x \le 255$
'\xFF'UC	`SlipDatum((UCHAR)'\xFF')`	255UC	uint8_t	$0 \le x \le 255$
1	`SlipDatum((LONG)1)`	1	int32_t	$--2^{(32-1)} \le x \le -2^{(32-1)} - 1$
1L	`SlipDatum((LONG)1)`	1	int32_t	$--2^{(32-1)} \le x \le -2^{(32-1)} - 1$
11	`SlipDatum((LONG)011)`	9	int32_t	$--2^{(32-1)} \le x \le -2^{(32-1)} - 1$
-1	`SlipDatum((LONG)-1)`	-1	int32_t	$--2^{(32-1)} \le x \le -2^{(32-1)} - 1$
0xFFFF	`SlipDatum((LONG)0xFFFF)`	-1	int32_t	$--2^{(32-1)} \le x \le -2^{(32-1)} - 1$
0xFFFFL	`SlipDatum((LONG)0xFFFF)`	-1	int32_t	$--2^{(32-1)} \le x \le -2^{(32-1)} - 1$
1U	`SlipDatum((ULONG)1)`	1U	uint32_t	$0 \le x \le -2^{(32)} - 1$
1UL	`SlipDatum((ULONG)1)`	1U	uint32_t	$0 \le x \le -2^{(32)} - 1$
011U	`SlipDatum((ULONG)011)`	9U	uint32_t	$0 \le x \le -2^{(32)} - 1$
0xFFFFU	`SlipDatum((ULONG)0xFFFF)`	4294967295U	uint32_t	$0 \le x \le -2^{(32)} - 1$

Legend

INPUT	The format in the input list file Where applicable, U and C and be upper or lower case.
Application	The format used to convert the input to the equivalent value in a list and the typical format for the same value in the application.
OUTPUT	The format of the output in the list file. Where applicable, U and C and be upper or lower case.
cstdint	The representation typedef equivalence for the application caste.
range	The range of integral values associated with the type.

Table 8.1.4-2: Floating Point Equivalences

INPUT	Application	OUTPUT	cfloat	range
1	`SlipDatum(1.)`	1	double	$--2.22507\ 10^{308} \le x \le 1.797693\ \ 10^{308}$
1.E0	`SlipDatum(1.E0)`	1	double	$--2.22507\ 10^{308} \le x \le 1.797693\ \ 10^{308}$
,1	`SlipDatum(.1)`	0.1	double	$--2.22507\ 10^{308} \le x \le 1.797693\ \ 10^{308}$
1E-1	`SlipDatum1E-1)`	0.1	double	$--2.22507\ 10^{308} \le x \le 1.797693\ \ 10^{308}$
1E-5	`SlipDatum(1E-5)`	0.00001	double	$--2.22507\ 10^{308} \le x \le 1.797693\ \ 10^{308}$
-1E+5	`SlipDatum(-11E+5)`	-10000	double	$--2.22507\ 10^{308} \le x \le 1.797693\ \ 10^{308}$

Legend

INPUT	The format in the input list file The exponent, 'E,' can be upper or lower case.
Application	The format used to convert the input to the equivalent value in a list and the typical format for the same value in the application.
OUTPUT	The format of the output in the list file. The exponent, 'E,' can be upper or lower case. Actual output formats depend on the precision and range of the input.
cfloat	The representation typedef equivalence for the application caste.
range	The range is specific to the range for gcc, see helpcentreonline.com/article/primitiv_console_gcc.pdf

Table 8.1.4-3 String Equivalencies

INPUT	Application	OUTPUT
“”	`SlipDatum(“”, false)`	“”
	`SlipDatum(“”, true)`	“”
“abc”	`SlipDatum(“abc”, false)`	“abc”
	`SlipDatum(“abc”, true)`	“abc”
“a\nb”	`SlipDatum(“a\nc”, false)`	a b
	`SlipDatum(“a\nc”, true)`	a b
“\x0A	`SlipDatum(“a\x0Ac”, false)`	a b
	`SlipDatum(“a\x0Ac”, true)`	a b

Legend

INPUT	The format in the input list file. There is no equivalent format to a constant string in the input. All input strings are considered to be non-constant. Changing the value in one string will not change the value in any other, identically appearing string.
Application	The format used to convert the input to the equivalent value in a list and the typical format for the same value in the application.
OUTPUT	The format of the output in the list file. There is no equivalent to a constant string on output. All output strings are considered non-constant.

8.2 List Language

The language defines constructs sufficient to guarantee round-trip processing of a list. The list output replicates features in the list, and the list input recaptures those features, restoring the list to be identical to its original. There is one exception. The constant string type is not recovered on input. All input strings are non-constant. Floating point number input/output is exact.

An important consideration in round-trip processing is the recognition of list sharing, that is, multiple references to the same list. This is supported using references to named lists. A named list is a list and has a temporary name. Creating and referencing a name shares a list.

The list language allows declarations to be shared in multiple files and included during list definition. Each file can contain only declarations and include statements. Any file containing the definition of the returned list will cause input processing to stop. Include cycles (recursion) is ont supported.

A list file can be output by SLIP or can be modified or constructed in a text editor. Both forms are readable by the SLIP reader. The only required content of a list file is the list definition.

A list file can contain the following elements:

comment	Inline (/* */) and line (//) comments are supported. Inline comments can be multiline. Line comments are only one line.
include directive	Include files can not contain a list definition. Include files can be nested (an include file can contain another include file). Include file names must be unique. All definitions in an include file are processed before resumption of processing at the next higher level.
user data list	A list of Application Data classes used in named lists or the list definition.
Application Data reference	A reference to a Application Data class in a named list or list definition accompanied by the object data. Causes the object data to be processed by the Application Data class parser and an Application Data SlipDatum object to be inserted into a list.
named list	Used to allow shared sublists to be created and/or to simplify the list definition. A named list is a list with a name. It has all the characteristics of a list definition except that it can not be returned as a list.
named list reference	A reference to a named list. The named list is copied into a descriptor list or becomes a shared list
descriptor list	Any list can have one and only one descriptor list. This is defined in the SlipHeader Section.
list definition	The list to be returned. It can contain sublists, shared lists (name lists), Application Data objects, and a descriptor list. The definition is recursive in that any sublist can have the same components. A sublist (or named list) is a list.

A simple example of a list file is:

```
user data1, data2;                          // Application Data class names
list1 ( 1 2U 3uc);                          // named list
list2 ( < {list1} “string” > 3.714 'a');    // named list w/descriptor list
list3 ( '\x0A');                            // named list

/*
 * This is the list returned to the user. It is the last
 * list processed. If this, the list definition, is seen
 * anywhere in the first list input file or in any include
 * file, list parsing will stop and this list will be
 * returned. The format of the named list and the list
 * definition is the same. A named list is a list with
 * a name.
*/

(“main string” {list1} {list2} {list1} ( {list3} data1(1c 2C) data2(3u 4uc))

/*
 * Input list parsing will not parse this comment. List
 * List processing stops as soon as the list definition
 * is seen.
 *
*/
```

Example 8.2-1 List File

Example 7.2-1 shows the following aspects of a list file:

- Application Data declaration: Declares the Application Data object in the list file. This is optional. Warning messages are given if the listed Application Data class names are not registered.
- Named lists. Named list define reusable lists. Named lists can be used anywhere a list is required. Named lists which are not used in the returned list are reported as being unused.
- Description List. A list containing a Description List using a named list is given in the list2 definition. The Description List uses a reference to a named list as the key in a <**key**, **value**> tuple.
- List definition. The returned list is indicated by it being a list without a name. It contains references to
    - Named lists. Several named lists are used in the list definition, and
    - Application Data object. Each Application Data object is prefixed by the object class name and followed by a list representing the object contents, and

- Reused lists ({list1}). Each reused list references the same list.

8.2.1 Include File Syntax

An include file allows named lists, user data lists, and other include files to be used. Include files are expanded at the point of detection and the include file contents are scoped to the outermost file (the initial file parsed by the user). If any include file contains a list definition which is not a named list, parsing will terminate and all files will be closed.

No include file can reference itself directly or indirectly. Each include file must be unique.

SYNTAX: **# include** “path/filename”
# include “filename”

include	is a keyword
path	is a valid path in the operating system
filename	is a valid filename in the operating system
#	is required
/	is a path/filename separator
“	is required to bracket the path, filename
	blanks before or after the hash symbol, #, and the include keyword are ignored
	blanks in the path or filename are operating system dependent.

```
# include “UserData”
# include “NamedList”
# include “ListDefinition”

// File: UserData
user data1, data2;

/// File: NamedList
hobbes ( 1 2 3)
calvin( 5 6 6)

// File: ListDefinition
( {data1(“User Data”) {calvin} { {calvin} hobbes(1.4 5, “Hobbes”) )
```

Example 8.2.1-1: Include Syntax

Example 7.2.1-1 illustrates the following concepts:

- Three include files serving different purpose.

- File UserData contains the Application Data classes used in the list. It is processed to completion and then closed and the outermost file entered.
- File NamedList contains named lists used in the list definition. It is processed to completion and then closed and the outermost file entered.
- File ListDefinition contains the the return list. It is processed to conclusion and then a return is made to the application.

Another example.

```
# include "UserData"
# include "ListDefinition"
# include "NamedList"

// File: UserData
user data1, data2;

/// File: NamedList
name1 ( 1 2 3)
calvin( 5 6 6)

// File: ListDefinition
( {data1("User Data") {name1} { {calvin} data2(1.4 5, "Hobbes") )
```

Example 8.2.1-2: Include Syntax Errors

Example 7.2.1-2 illustrates the following concepts:

- Three include files serving different purpose.
- File UserData contains the Application Data classes used in the list. It is processed to completion and then closed and the outermost file entered.
- File ListDefinition contains the the return list. It is processed to conclusion and then a return is made to the application.User Data Syntax. Errors will occur because the List Definition was processed before the “NamedList” file was processed.
- File NamedList contains named lists used in the list definition. This file is not processed. The List Definition occurs before the include statement for this file preventing file processing.

8.2.2 Application Data Syntax

Prior to input processing, each Application Data class used must be registered. If during input parsing, a reference to an Application Data class is seen but the class has not been registered, then an Application Data object will not be created and the returned list will be in error.

The syntax for an Application Data list is:

SYNTAX: **user** identifier [**,** identifier, …]**;**

user	is a keyword
identifier	must conform to identifier naming conventions
,	separator
;	mandatory, terminal semicolon
	blanks before or after the separator are ignored

The syntax for an Application Data reference is:

SYNTAX: identifier (list)

identifier	must conform to identifier naming conventions
(list **)**	any valid list
	blanks before or after the identifier are ignored
	blanks before or after the parentheses are ignored

```
user matrix, eigen. regresssion;
( matrix( ( (1.1 1.2) (2.1 2.2) ) interpolate(5.56, 17.852) )
```

Example 8.2.2-1: Application Data Syntax

Example 7.2.2-1 illustrates the following:

- Precondition: Application Data classes matrix and eigen are registered by the application. Application Data class interpolate was registered but not listed – this is not an error.
- A warning message is issued that Application Data class **regression** was not registered by the software.
- After the list definition is processed,, a warning error is issued that Application Data class **regression** was not used.
- Application Data class **interpolate** is processed without error, it was registered prior to processing.

- The data passed to the **matrix** parser for input processing is a list consisting of two sublist. The first sublist is (1.1 1.2) and the second sublist is (2.1 2.2). If the Application Data class **matrix** uses an n-dimensional C++ matrix to store data, then this is translated by the **matrix** parser to
  `double matrix[][2] = { {1.1, 1.2}, {2.1, 2.2} };`.
- The data passed to **interpolate** is the list (5.56, 17.852). If **interpolate** uses some method to interpolate a value between these values, then **interpolate** could translate the input to
  `double value = func(5.56, 17.852);`

Application Data processing is an application issue. The parser passes a single list to the registered parser for the input named class. The registered parser is responsible for converting the input list in a manner it finds suitable, and then deleting the input list as necessary.

8.2.3 Named List Syntax

A named list represents a defined list. The name can be used anywhere a sublist can be used and can be used as a Descriptor List. When used as a sublist, the list is referenced. When used as a Descriptor List, the list is copied. References are not unique. Several references to the same named list reference the same list. Descriptor Lists are unique. Several uses of the same named list as Descriptor Lists are all unique.

All named lists identifiers must be unique .In any list input file and in any include file reachable from the list file, each named list identifier definition can appear once. That is:

```
list1 ( … );        // and
list1 ( … );        // is illegal
```

The list definition supplied with the named list is a list. It has all the properties, and can contain any of the elements or objects, of a list. The list definition may have sublists, may reference other named lists, and each list may have a Descriptor List which is defined by a named list or may contain a named list as a **key** or **value** in the <**key**, **value**> tuple.

A named list may be used before it is defined. This is called a “forward reference”. Forward references are resolved at the time a definition is seen.. For example:

```
define ( {forward” );
forward( );
```

is legal. When 'forward' is defined, 'define' is completed.

No named list may reference itself directly or indirectly. That is, circular references are not allowed (and will go undetected). This prohibition includes Descriptor Lists. A Descriptor List which is a part of the named list definition or any sublist contained or reachable from the outer list of the definition can not reference itself.

A named list reference may be the list definition returned to the user.

The syntax for a named list is:

SYNTAX: identifier { list }**;**

identifier	must conform to identifier naming conventions
{ **list** }	a legal list
;	mandatory, terminal semicolon
	blanks before or after the identifier are ignored
	blanks before or after the curly braces ('{' '}') are ignored
	blanks before or after the semicolon are ignored

The syntax for a named list reference is:

SYNTAX: **{**identifier**}**

identifier	must conform to identifier naming conventions
{ }	required brackets
	blanks before or after the identifier are ignored
	blanks before or after the curly braces ('{' '}') are ignored

```
fuzzy ( );                // an empty list
logic ( {fuzzy} );        // a list w/a reference to an empty list
{logic}                   // logic is the return list definition
```

Example 8.2.3-1 Named List

Example 7.2.3-1 shows:

- An example of a named list which is an empty list (fuzzy).
- A sublist reference to an empty named list (logic).
- The named list is returned as the List Definition.

8.2.4 List Syntax

The list definition is recursive.

- A list is delimited by parenthesis, '**(**', '**)**'. Everything between the parentheses is part of a list.
- An empty list is defined as a left parentheses followed by a right parentheses '**( )**'
- A list can contain zero or more **SlipDatum** objects
- A list can contain zero or more sublists
- A sublist is a list or can be denoted by a named list, **{name}**.
- A list can contain zero or one Description Lists and must be the first entry in the list.

There is a unique list returned as a result of parsing the input file, the List Definition. The List Definition can either be a list, as described in the List Syntax, or a named list. This is the only place where a named list can occur outside of a list. That is, either a named list is in a list, sic. Description List or it is the List Definition to be returned to the application.

SYNTAX: **(listitem …)**

( )	list brackets
listitem	zero or more list items (lists, **SlipDatum** objects or named lists) blanks before or after the list brackets are ignored blanks before or after a **listitem** are ignored

```
Definition ( {forward} );// forward is a sublist in definition
forward    (  );         // an empty list
                         // a complex list (1 3.7 () () ('c' () )
list       ( 1 3.7 {forward} {forward} ('c' {forward} ) )
```

Example 8.2.4-1 Lists

8.2.4.1 Description List

A Description List is a list surrounded by the brackets '<.' and '>' with the exception

- That the list consists of 2-tuples, <**key**, **value**>, and
- **<** **{**name**}** **>** means that the named list **{**name**}** is the Description List contents and not a sublist.

A named list can also be an item of a <**key**, **value**> pair. That is:

`< {name1} 1 {name2} 2 >`

are two <**key**, **value**> pairs included in the Description List and not a (very) complex way of defining a Description List.

A Description List must have an even number of items, where zero is considered even.

SYNTAX: < **listitem listitem … >**

( )	list brackets
listitem	zero or more list item pairs (lists, **SlipDatum** objects or named lists) blanks before or after the list brackets are ignored blanks before or after a **listitem** are ignored

There is one anomaly in this definition. Since a Description List is recursively defined as a list (with different brackets), Description List can have a Description List (which can have a …). This is syntactically legal and will be processed correctly. But an embedded Description List has no access or processing support at runtime. In a word, ya' can create it but then so what, it is unusable.

```
list1  { 1 2 );                  //
dlist1 ( < {list} > {list1}   ); // ( < 1 2 > (1 2) )
dList2 ( < {dlist1} > {list1} ); // ( < < 1 2 > 1 2 > ( 1 2 ) )
dList3 ( < 3 4 > 5 6 );
defList( {dList2} {dList3} );
{defList}                        // what a mess!
```

Example 8.2.4-2: Description List

Example 7.2.4.1-1 illustrates several points:

- `dList1 < {list1> {list1}` > have two different uses of `{list1}` which are not shared. The instance of `{list1}` inside the Description List is a copy of list1. The `{list1}` outside of the Description List is a sublist reference to list1. Any changes made in this sublist are reflected to all other sublists referencing the same list.
- `dList2 ( < {dlist1} > {list1} )` uses the definition of `{dlist1}` as the definition of the Description List for `dList2`. This means that the entire definition for `{dlist1}` is copied into the Description List for `dList2`. `dlist1` loses its identity as being referenceable list and becomes a stand-alone Description List embedded in a Description List. `List1` contained in `dlist1` is copied into the Description List for `dList2` and becomes its <**key**, **value**> pair. The reference to `{list1}` in the list body becomes a reference to `list1` in its

definition. This reference is shared. Any change made to the sublist will cause a simultaneous change to, for example, the same sublist reference in `dlist1`.

- `Dlist3` has a simple Description List and two list items.
- `defList` has no Description list but references `{dList2}` and `{dList3}`, each of which has a Description List. The references to in `{dList2}` and `{dList2}` preserve their references to `{list1}`. Any change in `{list1}` on either path will affect both references. Any change to either Description List will be local to the containing list.

8.2.5 Parser Syntax Equations

```
Grammar

    0 $accept: sublistDeclarations_Definition $end

    1 sublistDeclarations_Definition: declarations listDefinition END
    2                               | declarations namedList END
    3                               | declarations
    4                               | listDefinition END
    5                               | namedList END

    6 declarations: declarations declarationItem
    7             | declarationItem
    8             | END

    9 declarationItem: LIST forwardReferenceList ';'
   10                | name listDefinition ';'
   11                | USER userDataList ';'
   12                | include

   13 forwardReferenceList: forwardReferenceList ',' name
   14                     | name

   15 listDefinition: '(' mark description list ')'
   16               | '(' mark description ')'
   17               | '(' description list ')'
   18               | '(' description ')'
   19               | '(' mark list ')'
   20               | '(' mark ')'
   21               | '(' list ')'
   22               | '(' ')'

   23 description: '<' description descriptionList '>'
   24            | '<' descriptionList '>'
   25            | '<' description '>'
   26            | '<' mark '>'
   27            | '<' '>'

   28 descriptionList: descriptionItemlist
   29                 | mark descriptionItemlist
   30                 | namedList

   31 descriptionItemlist:
                        descriptionItemlist descriptionElement
descriptionElement
   32                 | descriptionElement descriptionElement

   33 descriptionElement: namedList
   34                    | datum
   35                    | '(' listItemList ')'

   36 include: '#' INCLUDE datum

   37 userDataList: userDataList ',' name
   38              | name

   39 list: listItemList

   40 listItemList: listItemList listItem
   41              | listItem

   42 listItem: datum
   43          | namedList
   44          | userData
   45          | listDefinition

   46 mark: '{' number '}'

   47 namedList: '{' name '}'

   48 userData: name listDefinition

   49 datum: BOOL
   50      | CHAR
   51      | UCHAR
   52      | number
   53      | FLOAT
   54      | STRING

   55 number: INTEGER
   56       | UINTEGER
   57       | CHARS
   58       | CHARU

   59 name: NAME
```

8.3 Output

Output methods are recursive, with recursive depth directly proportional to the nesting depth of the output list.

Round-trip and non-round-trip output are supported. Round trip output makes use of all the language features described. Non-round-trip output does not support list sharing, named lists, or Application Data lists. It is primarily a means to pretty-print a list.

The output of non-round-trip output can be to either a file or a string. Round-trip output must be to a file.

Examples of use are given in Example 8.3-1: Output.

```
string fileame = "Test";
ofstream out;
out.open("Test", std::fstream::out);

SlipHeader* list = new SlipHeader();
SlipDatum*  listKey   = new SlipDatum((CHAR)'a');
SlipDatum*  listValue = new SlipDatum((LONG) 1);

list->create_dList();
list->put(*listKey, *listValue);

SlipHeader* sublist = new SlipHeader();
SlipDatum*  sublistKey   = new SlipDatum((CHAR)'b');
SlipDatum*  sublistValue = new SlipDatum((LONG)2);

sublist->enqueue((LONG) 5);
sublist->create_dList();
sublist->put(*sublistKey, *sublistValue);

list->enqueue((LONG) 3).enqueue(*sublist).enqueue(*sublist);

/*
 * OUTPUT
 * ( < 'a' 1 > 3 ( < 'b' 2 > 5 ) ( < 'b' 2 > 5 ) )
 *
 */

list->writeQuick();                        // output to cout
list->writeQuick(out);                     // output to a stream
list->writeQuick(filename);                // output to a file
string strList = list->writeToString();    // return a string

out << endl << “Round-trip output” << endl;

/*
 * OUTPUT
 * list1  ( 'a' 1 );
 * list2  ( < {list4} > 5 );
 * list3  ( < {list1} > 3 {list2} {list2} );
 * list4  ( 'b' 2 );
 * {list3}
 */

list->write();
list->write(ostream& out);
list->write(filename);
```

Example 8.3-1: Output

Some notes on Example 8.3-1: Output:

- Character output ('a', and 'c') does not have a size suffix ('C'). The default size and sign is CHAR.
- The method using a filename argument open and close the file. Streams are not opened or closed.
- If the stream is a stringstream then the behavior is the same as for writeToString() except the return is stored in a stream and not a string.
- Non-round-trip output does not show that lists are shared. Round-trip output shows sharing.
- Round-trip output shows:
  - list sharing (list2).
  - Description Lists are named lists (list1 and list4).

Not shown is Application Data Output. Application Data is displayed as *class-name (list)*. If instead of '3' in Example 8.3-1: Output we had a Application Data class named UDP which output the list '(“STRING” 3.7)' the output would have been:

```
( < 'a' 1 > UDP( "STRING" 3.7 ) ( < 'b' 2 > 5 ) ( < 'b' 2 > 5 ) ) and

list1  ( 'a' 1 );
list2  ( < {list4} > 5 );
list3  ( < {list1} > UDP( "STRING" 3.7 ) {list2} {list2} );
list4  ( 'b' 2 );
{list3}
```

8.4 Input from a List File

The reader is iterative (no recursion) but uses the heap.

The reader parses and input list file and returns a list. There are three steps in reader use:

- Instantiate the class SlipRead.
- Register all Application Data classes used n the input list file.
- Parse the input list file.

The list of Application Data classes must be known by the user. If the input list file was created by SLIP, all the required Application Data class names are provided. If the input file was created by an outside source, the the creator must provide this information procedurally or in the input list file.

If the debug version of SLIP is being used, the user can turn on debugging prior to parsing. There are several independent options. The options allow tracing of the parser reductions, the lexer tokens, and hash table activity. They can be used for problem isolation in the input list file (where does the problem occur and who is responsible). The legal values are:

- SlipRead::HASH Output Hash Table debug statements.
- SlipRead::INPUT
- SlipRead::LEXER Output lexer debug statements.
- SlipRead::PARSER Output Parser debug statements.
- SlipRead::ALL Output all debug statements.

Debug selection is:

```
  SlipRead::setDebugON(SlipRead::LEXER | ...);
```

where one or all the options can be missing. If all options are missing, the default is SlipRead::ALL.

The input list file content is given in Section 7.1 and Section 7.2.

Constructor for SlipRead are given in Table 7.4-1.

Table 8.4-1: SlipRead Constructors

Constructor	Description
SlipRead(int debugFlag)	Constructor for SlipRead class.. The default is debug is off when SLIP debug compilation is used.

Table 8.4-1: SlipRead Constructors

Constructor	Description
SlipRead(const int size, SlipDatum const userData[], int debugFlag)	Constructor for SlipRead class.. The default is debug is off when SLIP debug compilation is used. An array of Application Data class object is input and the Application Data classes registered.
SlipRead(const int size, SlipDatum * const userData[], int debugFlag)	Constructor for SlipRead class.. The default is debug is off when SLIP debug compilation is used. An array of pointers to Application Data class object is input and the Application Data classes registered.

Table 8.4-2: SlipRead Methods

return	((SlipRead&)X).method()	Description
int	getError()	Returns the number of errors found. Zero means no errors.
SlipHeader&	read(string filename)	Parse an input file given by filename.
bool	registerUserData(const SlipDatum&)	Register a (single) Application Data class
bool	registerUserData(int size, SlipDatum* userData)	Register an array of Application Data classes
bool	registerUserData(int size, SlipDatum userData[])	Register an array of Application Data classes
void	setDebugON(int debugFlag = INPUT)	Turn debug on, default SlipRead::INPUT

The 2-fold way to reading an input list file. Two examples are provided. The first shows how to parse an input list file containing no Application Data objects, and the second shows how to parse an input file containing a Application Data object.

```
SlipRead* read = new SlipRead();
SlipHeader& list = read->read(“filename”);
```

Example 8.4-1 Reading a File without User Data

The second example is a little harder.

```
SlipDatum* UserData = new SlipDatum(UserDataClass);

METHOD 1:
SlipRead* read = new SlipRead(*UserData);
SlipHeader& list = read->read(“filename”);
```

```
METHOD 2:
SlipRead* read = new SlipRead(*UserData);
read->registerUserData(*UserData);
SlipHeader& list = read->read(“filename”);
```

```
delete UserData;
```

Example 1: Reading a File with User Data

References

1. An Introduction to the List Processing Language SLIP
   Joseph Weizenbaum
   Programming Systems and Languages
   McGraw-Hill Book Company, 1967
2. Knotted Lists
   Joseph Weizenbaum
   Communications of the ACM V05N3, 1963
3. Symmetric List Processor
   Joseph Weizenbaum
   Communications of the ACM V06N9, 1963
4. On the Reference Counter method
   J. Harold McBeth
   Communications of the ACM V07N1, 1964
5. SLIP
   L. D. Yarbrough
   Communications of the ACM V07N1, 1964
6. Commenting on the Implementation Issues
   Joseph Weizenbaum
   Communications of the ACM V07N1, 1964
7. More on the Reference Counter Method of Erasing Lists
   Joseph Weizenbaum
   Communications of the ACM V07N1, 1964

8. A Comparison of List Processing Computer Languages COMIT, IPL, LISP, SLIP
   Daniel G. Bobrow, Bertram Raphael
   Communications of the ACM V07N4, 1964
9. More on SLIP
   Sanford Elkin
   Communication of the ACM V07N5, 1964
10. A Note on the Formation of Free Lists
   William M. Waite
   Communications of the ACM V07N8, 1964
11. The Implementation of SLIP
   Donald B. Russell
   Communications of the ACM V08N5, 1965
12. The Implementation of SLIP
   Joseph Weizenbaum
   Communications of the ACM V08N5, 1965
13. ELIZA – A Computer Program For the Study of Natural Language Communication Between Man And Machine
   Joseph Weizenbaum
   Communications of the ACM V09N1, 1966
14. AUTOMAST: Automatic Mathematical Analysis and Symbolic Translation
   William E. Ball, Robert I. Berns
   Communications of the ACM V09N8, 1966
15. An Efficient Machine-Independent Procedure for Garbage Collection of Various List Structures
   H. Schorr, W. M. Waite
   Communications of the ACM V10N8, 1967
16. Recovery of Reentrant List Structures in SLIP
   Joseph Weizenbaum
   Communications of the ACM V12N7, 1971
17. A List Set Generator
   Stuart C. Shapiro
   Communications of the ACM V13N12
18. List Tracing in systems Allowing Multiple Cell-Types
   Robert R. Fenichel
   Communications of the ACM V14N8, 1973
19. Multiprocessing Compactifying Garbage Collection
   Guy L. Steele Jr.
   Communications of the ACM V18N9, 1977

20. The Introduction of a Paging Technique into the Symmetric List Processor, SLIP
F. H. Sage, D. V. Smith
General Electric Company, Santa Barbara, CA

Appendix I Alphabetic Method List

Slip User's Manual

Appendix I Alphabetic Method List

The alphabetic list below summarizes all methods within SLIP. The provides information on the method names and return values, and where it is appropriate to use the method. The interpretation of the columns is as follows:

- **return**: Return value of the method.
  - SlipDatum&: SLIP defined class
  - SlipHeader&: SLIP defined class
  - SlipSublist&: SLIP defined class
  - Iterator&: This represents a reference to one of the valid iterators, SlipReader or SlipSequencer. The method is called using ***I.***method() and the return value is ***I&***.
    - SlipReader&: The method was called using ***(SlipReader&)X***.method() and returns SlipReader&.
    - SlipSequencer&: The method was called using ***(SlipSequencer&)X***.method() and returns SlipSequencer&.
- **method**: Method name and arguments. Where and argument is followed by an '=' then the argument has a default value and the application need not specify the argument.
- **SlipCell**: An 'X' in a column means that the method can be used with the specified class, as in X.method() where X is an object of the class. The following SLIP classes are represented:
  - S: SlipSublist class.
  - H: SlipHeader class.
  - D: SlipDatum class.
- **Iterator**: An 'X' in a column means that the method can be used with the specified iterator as in X.method() where X is an object of one of the following iterators. Note that where the method can apply to either a SlipDatum, SlipHeader, or SlipDatum cell the interpretation is that the method applies to a SlipDatum object. This corresponds to the reference returned from the iterator by coding currentCell(), as in currentCell().method().
  - SlipReader class.
  - SlipSequencer class.
- **Static**: The method is static, an object is not required to invoke the method. It can be called using a class specification, as in SlipDatum::method(), or with an object, as in X.method() where X is one of SlipDatum, SlipHeader, or SlipSublist. The table identifies which classes apply to the static method.

Table A.I-1: Alphabetic Method List

		SlipCell			Iterator		static
		S	H	D	R	S	
return	**method()**						
Iterator&	advanceLEL()				**X**	**X**	
Iterator&	advanceLER()				**X**	**X**	
Iterator&	advanceLNL()				**X**	**X**	
Iterator&	advanceLNR()				**X**	**X**	
Iterator&	advanceLWL()				**X**	**X**	
Iterator&	advanceLWR()				**X**	**X**	
Iterator&	advanceSEL()				**X**	**X**	
Iterator&	advanceSER()				**X**	**X**	
Iterator&	advanceSNL()				**X**	**X**	
Iterator&	advanceSNR()				**X**	**X**	
Iterator&	advanceSWL()				**X**	**X**	
Iterator&	advanceSWR()				**X**	**X**	
void	avslHistory(bool)	**X**	**X**	**X**			**X**
bool	containsKey(SlipCell&)		**X**		**X**		
bool	contains(SlipCell&)		**X**		**X**		
SlipHeader&	create_dList()		**X**		**X**		
SlipCell&	currentCell()				**X**	**X**	
SlipHeader&	currentList()				**X**		
SlipHeader&	delete_dList()		**X**		**X**		
bool	deleteAttributes(SlipCell&)		**X**		**X**		
Iterator&	deleteCell()				**X**	**X**	
SlipCell&	dequeue()		**X**		**X**		
string	dump()	**X**	**X**	**X**	**X**	**X**	

Table A.I-1: Alphabetic Method List

		SlipCell			Iterator		static
		S	**H**	**D**	**R**	**S**	
return	**method()**						
void	deleteSlip()	**X**	**X**	**X**			**X**
void	dumpDList()		**X**		**X**		
string	dumpLink()	**X**	**X**	**X**	**X**		**X**
void	dumpList()		**X**		**X**		
SlipCell&	enqueue(*bool*)		**X**		**X**		
SlipCell&	enqueue(*char*)		**X**		**X**		
SlipCell&	enqueue(*double*)		**X**		**X**		
SlipCell&	enqueue(*long*)		**X**		**X**		
SlipCell&	enqueue(PTR, const *void** operation=default))		**X**		**X**		
SlipDatum&	enqueue(SlipDatum&)		**X**		**X**		
SlipSublist&	enqueue(SlipHeader&)		**X**		**X**		
SlipSublist&	enqueue(SlipSublist&)		**X**		**X**		
SlipCell&	enqueue(*string**, *bool* constFlag=false)		**X**		**X**		
SlipCell&	enqueue(*string*&, *bool* constFlag=false)		**X**		**X**		
SlipCell&	enqueue(*unsigned char*)		**X**		**X**		
SlipCell&	enqueue(*unsigned long*)		**X**		**X**		
SlipHeader&	flush_dList()		**X**		**X**		
SlipHeader&	flush()		**X**		**X**		
SlipCell&	get(SlipCell&)		**X**		**X**		
SlipCell&	getBot()		**X**		**X**		
ClassType	getClassType()	**X**	**X**	**X**	**X**	**X**	
SlipCell*	getLeftLink()	**X**	**X**	**X**	**X**	**X**	
uShort	getMark()		**X**		**X**		
*string**	getName()	**X**	**X**	**X**	**X**	**X**	
uShort	getRefCount()	**X**	**X**		**X**		

Table A.I-1: Alphabetic Method List

		SlipCell			**Iterator**		**static**
		S	**H**	**D**	**R**	**S**	
return	**method()**						
SlipCell*	getRightLink()	**X**	**X**	**X**	**X**	**X**	
SlipState	getSlipState()	**X**	**X**	**X**			**X**
SlipCell&	getTop()		**X**		**X**		
SlipDatum&	insLeft(bool)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(char)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(const PTR, const *void** operation=default)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(const *string**, *bool* constFlag=false)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(const *string*&, *bool* constFlag=false)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(double)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(long)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(SlipDatum&)	**X**	**X**	**X**	**X**	**X**	
SlipSublist&	insLeft(SlipHeader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	insLeft(SlipReader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	insLeft(SlipSequencer&)	**X**	**X**	**X**	**X**	**X**	
SlipSublist&	insLeft(SlipSublist&)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(*unsigned char*)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insLeft(*unsigned long*)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight(*bool*)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight(*char*)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight(const PTR, const *void** operation=default)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight(const *string**, *bool* constFlag=false)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight(const *string*&, *bool* constFlag=false)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight(*double*)	**X**	**X**	**X**	**X**	**X**	

Table A.I-1: Alphabetic Method List

		SlipCell			Iterator		static
		S	**H**	**D**	**R**	**S**	
return	**method()**						
SlipDatum&	insRight(*long*)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	insRight(SlipDatum&)	**X**	**X**	**X**	**X**	**X**	
SlipSublist&	insRight(SlipHeader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	insRight(SlipReader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	insRight(SlipSequencer&)	**X**	**X**	**X**	**X**	**X**	
SlipSublist&	insRight(SlipSublist&)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight(*unsigned char*)	**X**	**X**	**X**	**X**	**X**	
SlipDatum&	insRight*(unsigned long*)	**X**	**X**	**X**	**X**	**X**	
bool	isAVSL(const SlipCellBase*)	**X**	**X**	**X**			**X**
bool	isData()	**X**	**X**	**X**	**X**	**X**	
bool	isDeleted()	**X**	**X**	**X**	**X**	**X**	
bool	isDiscrete()	**X**	**X**	**X**	**X**	**X**	
bool	isEmpty()	**X**	**X**	**X**	**X**		
bool	isEqual(SlipHeader&)		**X**		**X**		
bool	isHeader()	**X**	**X**	**X**	**X**	**X**	
bool	isName()	**X**	**X**	**X**	**X**	**X**	
bool	isNumber()	**X**	**X**	**X**	**X**	**X**	
bool	isPtr()	**X**	**X**	**X**	**X**	**X**	
bool	isReal()	**X**	**X**	**X**	**X**	**X**	
bool	isString()	**X**	**X**	**X**	**X**	**X**	
bool	isSublist()	**X**	**X**	**X**	**X**	**X**	
bool	isTemp()	**X**	**X**	**X**	**X**	**X**	
bool	isUnlinked()	**X**	**X**	**X**			
short	listDepth()				**X**		
SlipCell&	moveLeft(SlipCell&)	**X**	**X**	**X**	**X**	**X**	

Table A.I-1: Alphabetic Method List

		SlipCell			**Iterator**		**static**
		S	**H**	**D**	**R**	**S**	
return	**method()**						
SlipCell&	moveLeft(SlipReader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveLeft(SlipSequencer&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListLeft(SlipHeader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListLeft(SlipReader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListLeft(SlipSequencer&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListLeft(SlipSublist&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListRight(SlipHeader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListRight(SlipReader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListRight(SlipSequencer&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveListRight(SlipSublist&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveRight(SlipCell&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveRight(SlipReader&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	moveRight(SlipSequencer&)	**X**	**X**	**X**	**X**	**X**	
SlipCell&	pop()		**X**		**X**		
void	printAVSL(*string*)	**X**	**X**	**X**			**X**
void	printClassSizes()	**X**	**X**	**X**			**X**
void	printDList()		**X**		**X**		
void	printFragmentList(*string*)	**X**	**X**	**X**			**X**
void	printList()		**X**		**X**		
void	printList(ostream&)		**X**		**X**		
void	printMemory(*string*)	**X**	**X**	**X**			**X**
void	printState(*string*)	**X**	**X**	**X**			**X**
SlipCell&	push(bool)		**X**		**X**		
SlipCell&	push(*char*)		**X**		**X**		
SlipCell&	push(*double*)		**X**		**X**		

Table A.I-1: Alphabetic Method List

		SlipCell			Iterator		static
		S	**H**	**D**	**R**	**S**	
return	**method()**						
SlipCell&	push(_long_)		**X**		**X**		
SlipCell&	push(PTR, const _void_* operation=default))		**X**		**X**		
SlipDatum&	push(SlipDatum&)		**X**		**X**		
SlipSublist&	push(SlipHeader&)		**X**		**X**		
SlipSublist&	push(SlipSublist&)		**X**		**X**		
SlipCell&	push(_string_*, _bool_ constFlag=false)		**X**		**X**		
SlipCell&	push(_string_&, _bool_ constFlag=false)		**X**		**X**		
SlipCell&	push(_unsigned char_)		**X**		**X**		
SlipCell&	push(_unsigned long)_		**X**		**X**		
SlipCell&	put(SlipCell&,SlipCell&)		**X**		**X**		
uShort	putMark()		**X**		**X**		
SlipDatum&	replace(_bool_)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(_char_)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(const PTR, const _void_* operation=null)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(const SlipDatum&)	**X**		**X**	**X**	**X**	
SlipSublist&	replace(const SlipHeader&)	**X**		**X**	**X**	**X**	
SlipSublist&	replace(const SlipSublist&)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(const _string_*, _bool_ constFlag=false)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(const _string_&, _bool_ constFlag=false)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(_double_)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(_long_)	**X**		**X**	**X**	**X**	
SlipCell&	replace(SlipReader&)	**X**		**X**	**X**	**X**	
SlipCell&	replace(SlipSequencer&)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(_unsigned char_)	**X**		**X**	**X**	**X**	
SlipDatum&	replace(_unsigned long_)	**X**		**X**	**X**	**X**	

Table A.I-1: Alphabetic Method List

		SlipCell			Iterator		static
		S	**H**	**D**	**R**	**S**	
return	**method()**						
SlipDatum&	replaceBot(*bool*)		**X**		**X**		
SlipDatum&	replaceBot(*char*)		**X**		**X**		
SlipDatum&	replaceBot(const PTR, const *void** operation=null)		**X**		**X**		
SlipDatum&	replaceBot(const SlipDatum&)		**X**		**X**		
SlipSublist&	replaceBot(const SlipHeader&)		**X**		**X**		
SlipSublist&	replaceBot(const SlipSublist&)		**X**		**X**		
SlipDatum&	replaceBot(const *string**, *bool* constFlag=false)		**X**		**X**		
SlipDatum&	replaceBot(const *string*&, *bool* constFlag=false)		**X**		**X**		
SlipDatum&	replaceBot(*double*)		**X**		**X**		
SlipDatum&	replaceBot(*long*)		**X**		**X**		
SlipDatum&	replaceBot(*unsigned char*)		**X**		**X**		
SlipDatum&	replaceBot(*unsigned long*)		**X**		**X**		
SlipDatum&	replaceTop(*bool*)		**X**		**X**		
SlipDatum&	replaceTop(*char*)		**X**		**X**		
SlipDatum&	replaceTop(const PTR, const *void** operation=null)		**X**		**X**		
SlipDatum&	replaceTop(const SlipDatum&)		**X**		**X**		
SlipSublist&	replaceTop(const SlipHeader&)		**X**		**X**		
SlipSublist&	replaceTop(const SlipSublist&)		**X**		**X**		
SlipDatum&	replaceTop(const *string**, *bool* constFlag=false)		**X**		**X**		
SlipDatum&	replaceTop(const *string*&, *bool* constFlag=false)		**X**		**X**		
SlipDatum&	replaceTop(*double*)		**X**		**X**		
SlipDatum&	replaceTop(*long*)		**X**		**X**		
SlipDatum&	replaceTop(*unsigned char*)		**X**		**X**		

Table A.I-1: Alphabetic Method List

		SlipCell			**Iterator**		**static**
		S	**H**	**D**	**R**	**S**	
return	**method()**						
SlipDatum&	replaceTop(*unsigned long*)		**X**		**X**		
SlipHeader&	reset()					**X**	
SlipSequencer&	reset(SlipCell&)				**X**		
SlipSequencer&	reset(SlipHeader&)				**X**		
SlipSequencer&	reset(SlipReader&)				**X**		
SlipSequencer&	reset(SlipSequencer&)				**X**		
SlipReader&	resetTop()				**X**		
unsigned	size_dList()		**X**		**X**		
unsigned	size()		**X**		**X**		
void	slipInit(*long*, *long*)	**X**	**X**	**X**			**X**
SlipHeader&	splitLeft(SlipCell&)		**X**		**X**		
SlipHeader&	splitRight(SlipCell&)		**X**		**X**		
void	sysInfo(ostream&)	**X**	**X**	**X**			**X**
string	toString()	**X**	**X**	**X**	**X**	**X**	**X**
SlipCell&	unlink()	**X**		**X**	**X**	**X**	
SlipReader&	upLevel()				**X**		
void	write()		**X**		**X**		
void	write(*string*&)		**X**		**X**		
void	write(*ostream*&)		**X**		**X**		
void	writeQuick()		**X**		**X**		
void	writeQuick(*string*&)		**X**		**X**		
void	writeQuick(*ostream*&)		**X**		**X**		
string	writeToString()		**X**		**X**		

Table A.I-2: SlipReader Alphabetic Method List

return	**((SlipRead&)X).method()**	**Description**
int	getError()	
SlipHeader&	read(string filename)	Parse and input file given by filename.
bool	registerUserData(const SlipDatum&)	Register a (single) Application Data class
bool	registerUserData(int size, SlipDatum* userData[])	Register an array of Application Data classes
bool	registerUserData(int size, SlipDatum userData[])	Register an array of Application Data classes
void	setDebugON(int debugFlag = INPUT)	Turn debug on, default SlipRead::INPUT

Appendix II SlipCell Dump Styles

Dumps are formatted versions of specific information contained in a SlipCell. They consist of a common prefix, generated by dumpLink(). The remainder of the dump format is generated by dumps for given SlipCells, SlipHeader.dump() on page 57, SlipSublist.dump() on page 66, and SlipDatum.dump() on page 67. The SlipDatum dumps are granulated into dumps for bool, CHAR, UCHAR, LONG, ULONG, DOUBLE, string and PTR. Each of these granulated dumps have outputs characteristic of the individual types with the notation that the PTR dump is defined by the application and not SLIP.

SLIP operational information is included in the output and contains specific pointer values and representation of internal operating data. Access to this information is restricted with limited access through method calls.

SlipCell link pointers can have the following values:

1. hexadecimal value: The location of the cell in memory. All locations are designated by "0x" with the number of hexadecimal determined by the operating system SLIP was compiled for (32-bit of 64-bit).
2. NULL: The SlipSublist cell is not on a list. Both SlipCell pointers will be **null**.
3. TEMPORARY: The SlipDatum cell is on the runtime stack. Both SlipCell pointers will be TEMPORARY.
4. DEADBEEF: The SlipDatum cell is on the AVSL. The left link will be 0xDEADBEEF and the right link will be a hexadecimal value – a pointer to another SlipCell on the AVSL – or NULL, indicated that the current cell is the last one in the AVSL.

A description of each format is given below along with examples.

```
[ type ] address::<pointer pointer> [operator pointer]

[ bool ] 0x22a820::<TEMPORARY TEMPORARY> [operator pointer]
```

Figure A.II-1: Common Dump Prefix

The operator pointer points to a static class object which performs type specific operations. Each SlipCell participates with the same set of operations but in a type specific way, and has the same properties by the values are specific to the type. The SlipCell characteristics tailored to the type is given in the static object pointed to by the operator pointer. This is an internal SLIP interface not available to the application, except for PTR types, see SlipDatum(const PTR, void* operation=null).

The address field will always contain a valid memory address.

The valid type fields are:

SlipHeader	SlipHeader()
SlipSublist	SlipSublist(new SlipHeader())
bool	SlipDatum((bool)X)
char	SlipDatum((CHAR)X)
uchar	SlipDatum((UCHAR)X)
long	SlipDatum((LONG)X)
ulong	SlipDatum((ULONG)X)
double	SlipDatum((DOUBLE)X)
	SlipDatum((PTR)X) value determined by the application

A SlipHeader dumps internal information on the number of SlipSublists referencing the current list (RefCnt), the value of an application data mark (mrk:), and a pointer to a descriptor list or **null**.

A full dump for a SlipHeader object on a 64 bit system is shown in Figure A.II-2: SlipHeader Dump Format. The common prefix is shown followed by the specific SlipHeader field values for the reference count, the mark and and the descriptor pointer. The list is empty (the left and right link pointers point to the SlipHeader object) and the list is referenced in one other list (RefCnt is 1). The application has not applied a application specific data value (mrk is 0) and there is not descriptor object.

```
[header   ] 0x600060cf0::<0x600060cf0 0x600060cf0> [0x600010dd0]  RefCnt: 1 mrk: 0
Descriptor NULL
```

Figure A.II-2: SlipHeader Dump Format

The SlipSublist dumps the common dump prefix followed by a pointer (→) to the SlipHeader dump of the referenced list. A SlipSublist object must be associated with a SlipHeader.

```
[sublist  ] 0x600060e70::<NULL        NULL       > [0x600010fb0]
-> [header   ] 0x600060cf0::<0x600060cf0 0x600060cf0> [0x600010dd0]  RefCnt: 1 mrk: 0
Descriptor NULL
```

Figure A.II-3: SlipSublist Dump Format

Figure A.II-3: SlipSublist Dump Format shows that the object is not on a list (the left and right link pointers in the common prefix are null) and that the list pointed to is empty (the left and right link pointers point to the address of the SlipHeader object). By comparing the list object address we see that the list pointed to is the same one shown in Figure A.II-2: SlipHeader Dump Format.

The format for SlipDatum cells are the same, but the output value depends on the type of cell being dumped. Each type has a different output. See Figure A.II-4: SlipDatum Dump Format .

```
[bool     ] 0x22a820::<TEMPORARY TEMPORARY> > [0x600010ca0]  = true
[char     ] 0x22a850::<TEMPORARY TEMPORARY> > [0x600010d00]  = 'A'
[uchar    ] 0x22a880::<TEMPORARY TEMPORARY> > [0x600011060]  = 'B'
[long     ] 0x22a8b0::<TEMPORARY TEMPORARY> > [0x600010e20]  = 4660
[ulong    ] 0x22a8e0::<TEMPORARY TEMPORARY> > [0x6000110c0]  = 9320
[double   ] 0x22a910::<TEMPORARY TEMPORARY> > [0x600010d60]  = 1.2340000
```

Figure A.II-4: SlipDatum Dump Format

We see that all the SlipDatum cells are on the runtime stack (the SlipDatum address is very different from the operation pointer address – which comes from the heap) and that the common dump prefix is suffixed by a '='. The type/values are given in Table A.II-1: SlipDatum Dump Values.

Table A.II-1: SlipDatum Dump Values

type	**value**
bool	**true** or **false**
CHAR	Hex for non-graphic ASCII, otherwise the graphic character
UCHAR	Hex for non-graphic ASCII, otherwise the graphic character
LONG	Integer
ULONG	Integer
DOUBLE	Formatted floating point
PTR	Application dependent

Copying

FDL 1.3

1 .0 GNU Free Documentation License (FDL 1.3)

The **GNU Free Documentation License (FDL 1.3)** or later version is acceptable. The **FDL 1.3** version is given below.

1.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

1.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

1.4 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

1.5 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

1.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

1.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

1.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

1.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

1.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

1.11 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

1.12 RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.