SLIP

Symbolic List Processor

User's Manual

2026

Table of Contents

1.0 Introduction

SLIP is an Application Programming Interface (API) supporting the creation and manipulation of lists of list, that is, a list which can reference a list. In other contexts this can be viewed as a graph where each node on the graph is a list. Both terms will be used as appropriate. The data carrying objects can be one of several known data types and a participate in arithmetic, logical, bit, or string operations as appropriate.

The current SLIP itself is an extension of the J. Weizenbaum's 1964 Communications of the ACM (CACM v6n9) article. In that article Weizenbaum described a Symmetric List Processor which supported a list of lists, and mechanisms to create, delete and iterate over the list. The original SLIP was written in FORTRAN II, and in 1966 in MAD for ELIZA (CACM V9N1), the worlds first chatbot.

This current implementation uses the 1964 SLIP as a basis. It deletes functions which are archaic, uses function names appropriate to current Computer Science terminology and expands SLIP functionality

Treating the lists of lists as a graph, we can say that SLIP supports a Directed Cyclic Graph (DAG). A Directed Cyclic Graph (DG), in which a list refers to itself, produces a Graph with unrecoverable nodes (lists) in SLIP. A DAG does not have any cycles. Hence if DGs were allowed the resultant structure would produce memory holes, using a DAG avoids memory holes.

SLIP contains four data structures. There is the list header , SlipHeader, which contains two lists; a Description List of <key, value> pairs, and a symmetric list. The Description List and symmetric list contain SlipDatum objects which are the repositories of data. There is a list iterator, SlipReader, which functions as a list iterator in a similar sense as the C++ Standard Template Library (STL) iterators, and supports operations common to that available for lists (SlipHeader objects) and data (SlipDaum objects). And there is a separate Input/Output (I/O) facility.

Concessions in the implementation have been made towards performance. The implementation is designed to minimize runtime performance.

1.1 Glossary

`cell`	Either `data` or a SlipHeader object
`data`	Primitive data input, C++ types bool, char, long int, double, string, unsigned char, unsigned long int
dList	SlipHeader Descriptor List – a list of <key, value> SlipDatum objects
`font`	Shell commands and C++ code
Italics	
Iterator	SlipReader
list	SlipHeader list
refcnt	SlipHeader Reference Count, number of SlipDatum sublists referencing the current list
SlipCell	Either the SLIP SlipDatum or SlipHeader classes
SlipDatum	Container for `data` and SlipHeader references (sublist)
SlipHeader	SLIP list unique header
SlipReader	Slip iterator allows traversal of lists and sublists and access to SlipDatum objects
sublist	SlipDatum object referencing a SlipHeader object
topmost	The list 'root' in a list of lists. In a list of lists, this is the first list.

1.2 Compiling and Installing

The compilation instructions are based on implementation in a Cygwin or Linux environment. The transition to other systems depends on the requirements of those systems, the requirements given here should apply equally to a new environment.

The instructions show the creation of a Dynamic Link Library (DLL) and a Static Library. The application is expected to link to the appropriate library during executable creation.

1.2.1 Cygwin

Cygwin is a Linux system re-hosted to be Windows compliant. It can be downloaded and installed from https://cygwin.com/ Installation instructions are included in the website. A synopsis is (1) create a space for the downloaded data, (2) create a space for the operational software, (3) download the setup

executable, (4) execute the setup executable, (5) change the Windows Environment PATH variable if you are going to execute the 'Linux; commands in a Windows command shell. Thereafter, the Linux commands are available in the Cygwin (bash) shell or in the Windows 'cmd' shell.

Linux contains two parts; (a) a kernel, and (b) applications. For Cygwin, Windows acts as the kernel. Cygwin interposes a dll between the Linux applications and Windows, to ensure that Linux requirements are satisfied. When working within Cygwin you are using Linux. It is assumed that the Cygwin commands will work equally well in a Linux environment.

The objective of this section is to compile the API using Linux commands using the Cygwin terminal. To do this (without make) do:

```
> mkdir -p <path>/SLIP/obj
> cp <from SLIP path>* <path>/SLIP/
> cd <path>/SLIP
> for I in *.cpp;do g++ -c -O2 -std=c++20 -o obj/${i%%.cpp}.o $i;done
```

'`mkdir`' creates the folder for the source code and compiled (object) modules. '`cp`' copies the source code to the source code directory, `<path>SLIP`, '`cd`' transfers to the source code directory. '`for`' compiles the code using optimization level '`O2`" and the C++ 2020, `C++20`, standard. The resultant object files are placed into `<path>SLIP/obj`, '`${i%%.cpp}.o` changes the source file being compile, *basename.cpp*, to the object file format, *basename.o.*

1.2.2 Linking and Installation

The object code is now linked into a '`.dll'` and a static library, '`.a`', The `dll` and static library are then copied to a common space that the applications can link to the software curing application linking. For our purposes, this command space is located at /usr/local/bin, which is part of the Windows environment PATH and the Cygwin PATH.

```
> cd <path>/SLIP/obj
> g++ -shared -o cygslip.dll \
      -Wl,--out-implib=libslip.dll.a \
      -Wl,--export-all-symbols \
      -Wl,--enable-auto-import *.o;

  > cp -u cygslip.dll /usr/local/lib;
  > cp -u libslip.dll.a /usr/local/lib;
```

'cd' changes the directory to the directory containing the object (.o) files. The '`g++`' command and options are the linker commands required to create DLL (`cygslip.dll`) file and a Static Library (`libslip.dll.a`) files. The terminal '`\`' at the end of the line are Linux continue line endings. If '`\`' is the end of a command line in a Linux shell, and is followed by and end-of-line, then the next line is considered part of the same command. Here the command is contained on 4 lines.

The 'cp' commands copy the .dll file and the .a file to a communal location available to applications during linking. Here we have selected `/usr/local/lib` as that communal location.

1.2.3 Include Files

SLIP include files (`.h`) are copied into a folder communally accessible during application compiling. The communal path selected (which can be changed) is:

```
> mkdir -p /usr/local/include/slip/
> cp <path>/SLIP/*.h /usr/local/include/slip/
```

Where the '`mkdir`' creates a path to the communal location of the SLIP header files, and '`cp`' copies the header files. This path must be known to the compilation command line compiler (`g++` for Cygwin) if a command line compiler is used.

2.0 Getting Started

This section is a discussion of the application environment. How does the application start using the SLIP API.

2.1 API Interface

Application using SLIP functions should include:

```
# include <slip.h>
```

in the C++ header or source file.

2.1.1 SlipInit

SLIP requires a private memory heap for it's operation. It does not use the general C++ heap except in acquiring space. The SLIP heap is a contiguous space. This space is partitioned into a SLIP list with each member of the list a SlipCell. This space is called the Available Space List (AVSL).

All SLIP cells are the same size. For this reason, the AVSL has no 'holes', memory space that is no longer usable because it is too small. All SLIP cells are retrieved from the AVSL using `new` and deleted using `delete`.

The application has two courses of action. The application can do nothing, in this case at first use of '`new`' for a SLIP cell, space is created without further application action.

The initial heap is used until exhausted. At that time, additional space is gotten from the C++ general heap and linked into the AVSL. This will continue indefinitely. As space is consumed and more space is needed, it is acquired from the C++ general heap.

If the application requires that the initial heap size does not change then `slipInit()` must be used. If `slipInit()` is called before the first SLIP related '`new`', then the initial space allocation used is the user provided allocation. If `slipInit()` is used after the first SLIP '`new`', then allocation of additional space will use the modified AVSL acquisition value.

Applications operating in a space constrained environment can use `slipInit()`. If space is exhausted and additional space is required a catachable SLIP exception is thrown.

The format of the `slipInit()` is:

```
slipInit(unsigned long allocation, unsigned long delta);
```

The `allocation` argument specifies the amount of AVSL space to be allocated initially. The `delta` argument specifies the amount of AVSL space to be allocated when the current AVSL space is exhausted.

The sizes provided as arguments specifies the number of SLIP cells to allocate. Each SLIP cell is 48 bytes. The default `allocation` is 1,000 SLIP cells, or 48,000 bytes in a 64-bit computer. When the current space is exhausted an additional `delta` allocation will be allocated. If `delta = 0` then no data will be allocated and a SlipException will be thrown.

2.1.2 Data Types

A SLIP data object is a heterogeneous data carrier. The allowed C++ Table 2.1 shows the valid `data` types.

Table 2.1
SLIP `Data` Types

C++ Data Type	values
bool	true, false
char	ASCII 8-bit signed character
unsigned char	ASCII 8-bit unsigned character
int32_t	32-bit signed integer
uint_32_t	32-bit unsigned integer
double	Double precision floating point
string	C++ defined string

3.0 SLIP Structures

A list is a structure consisting of a list header (SlipHeader) and list data cells (SlipDatum). The list header and the data cells are bidirectional. There are two pointers in each cell; one pointing to the preceding cell and one pointing to the next cell. A list cell can point to a sublist, allowing a list of lists to be constructed. A sublist has a pointer to its descendant list. See Figure 3.1.

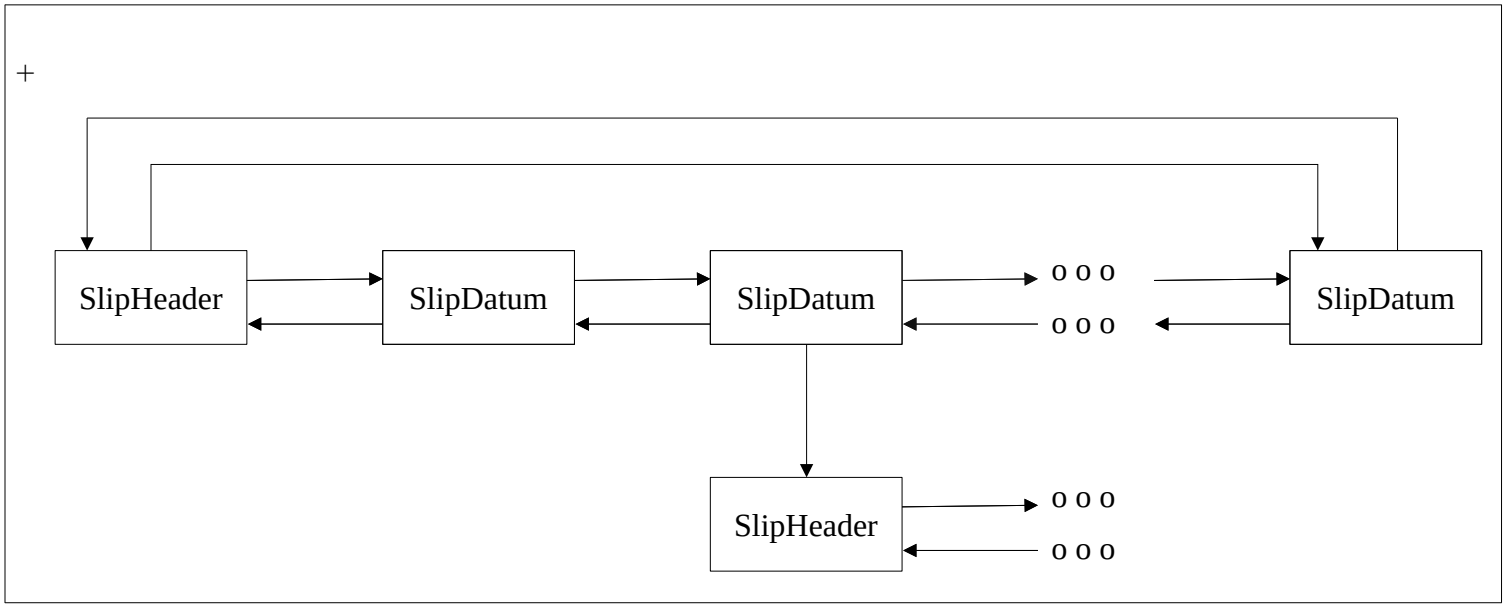


Figure 3.1
SLIP List Structure

Figure 3.1 shows two lists. The first one has a list header (SlipHeader) and a number of list data cells (SlipDatum), the second of which points to another list SlipHeader. This is a list of lists. If we consider this as a graph, then the first list is the graph root, and the second list is a subgraph

Traversing this list is done using a SLIP iterator (SlipReader). The name 'SlipReader' is an historical reference to a similarly named iterator in the original SLIP. However in current terminology, this would be called an iterator with iterator semantics. An iterator is used to descend into a list (a contained list) and to ascend from the contained list back to its origin. Ascending from a list can only be done with an iterator. A contained list does not have a pointer to its container, and a list can be contained in multiple lists.

In summary, a list consists of a SlipHeader SLIP cell and zero or more SlipDatum SLIP cells. A SlipDatum cell can contain `data` or a reference to a contained list. Traversing a list is done using an iterator.

3.1 SlipHeader

The list header (SlipHeader) contains information concerning the list. This includes:

1. A pointer to the list top and list bottom.
2. A Descriptor List (dList). A dList is a Description List with <key, value> SlipDatum pairs.
3. A user Mark which allows a user the ability to tag the list with a 15-bit numerical value [0 – 32,767].
4. A Reference Count (refcnt), the number of lists that the current list is contained in. A given list can have no more than 65,535 sublist references.

   SlipDatum sublist deletion reduces the Reference Count by 1. When the Reference Count is 0, the list is returned to the AVSL.

Figure 3.2 provides a SlipHeader overview. There is a single pointer to the dList, and a doubly linked pointer to the list. The dList is constructed the same as a SLIP list, that is, the dList is a list with a relation between list pairs.

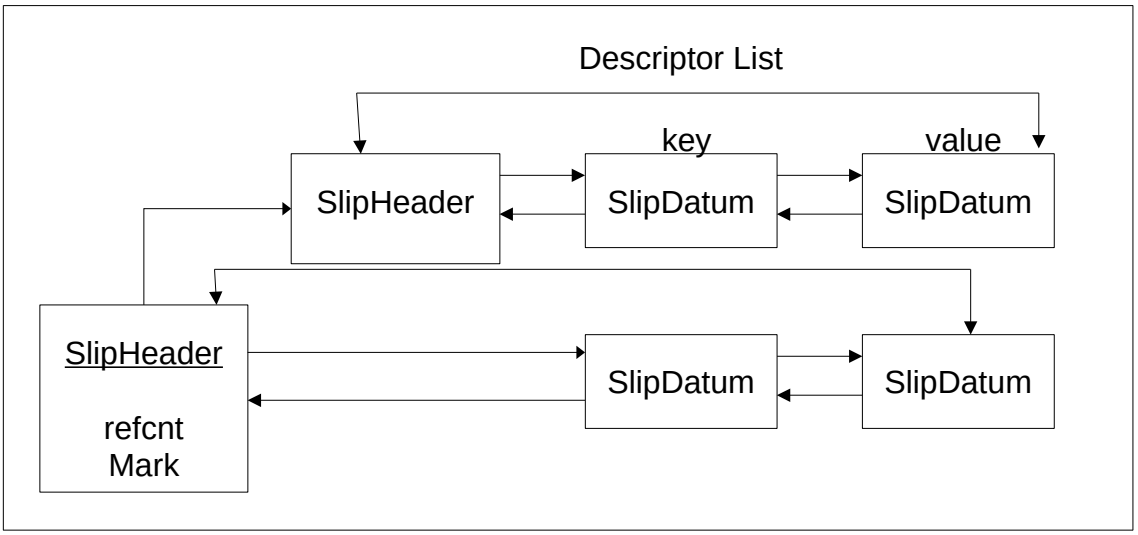


Figure 3.2
SlipHeader Structure

3.2 SlipDatum

The SlipDatum cell is a data carrier. It provides the sugar rquired to carry all of the data types defined in Section 2.1.2 Data Types plus a reference to a list, called a 'sublist. Examples of the creation of a SlipDatum object are given in Table 3.1.

A SlipDatum object eferencing SlipHeader object is a sublist. It provides the SLIP mechanism to support a list of lists.

Table 3.1
SLIP `cell` Types

`cell`	Return value	new SlipDatum()
bool	SlipDatum*	SlipDatum(true)
char	SlipDatum*	SlipDatum('a')
unsigned char	SlipDatum*	SlipDatum((unsigned char) 'b')
int32_t	SlipDatum*	SlipDatum(1)
uint_32_t	SlipDatum*	SlipDatum((unsigned) 2)
double	SlipDatum*	SlipDatum(0.5772)
SlipDatum	SlipDatum*	SlipDatum(datum)
sublist	SlipDatum*	SlipDatum(new SlipHeader)

Except for strings, constant values are copied and placed directly into a SlipDatum object. For strings, a new string is created and a reference to the string is placed into the SlipDatum object. Space for a string is acquired from the heap. On deletion of the SlipDatum object or on substitution (*object* = *value*) the space is released to the heap.

A SlipDatum object argument causes a copy of the SlipDatum contents into a new object. If the input is a sublist, then the referenced list SlipHeader refcnt is incremented.

A SlipHeader object argument causes the creation of a sublist referenct to the SlipHeader and and the SlipHeader refcnt is incremented.

Variables are treated in the same way. The variable value is copied into the new object. If the input is a sublist, then the SlipHeader refcnt is incremented.

Creation of a SlipDatum cell can be explict or implied. Functions which can create a SlipDatum object as part of their operation will be called "implied". For example, new SlipDatum(`cell`) is explicit, but insRight(`cell`) is implicit.

Each data type has operations that can be performed on them. The arithemetic and boolean types can do arithmetic, logical, and cast operations. The numerical types can also do shift operations. The sublist type can do operations specific to a SlipHeader when using the contained SlipHeader reference.

3.3 SlipReader

This is the SLIP iterator. Using 'Reader' is a relic of the initial SLIP where Weizenbaum had two iterators, a *sequencer* and a *reader.* The sequencer has been abandoned. The *reader* remains and the class name is SlipReader.

The iterator supports traversal across a list, either linearly across a single list, or structurally 'entering' and 'exiting' contained lists. Effectively, the iterator has a memory and remembers at which level of a list it is in, and uses this memory to exit a subordinate list to return the the containing list. A diagrammatic view of the internals can be seen in Figure 3.3.

Figure 3.3
Iterator Partition

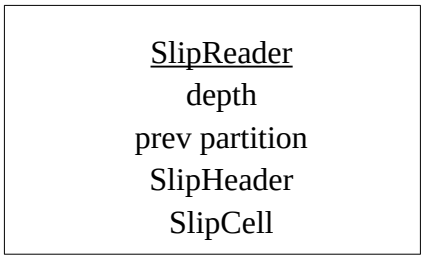
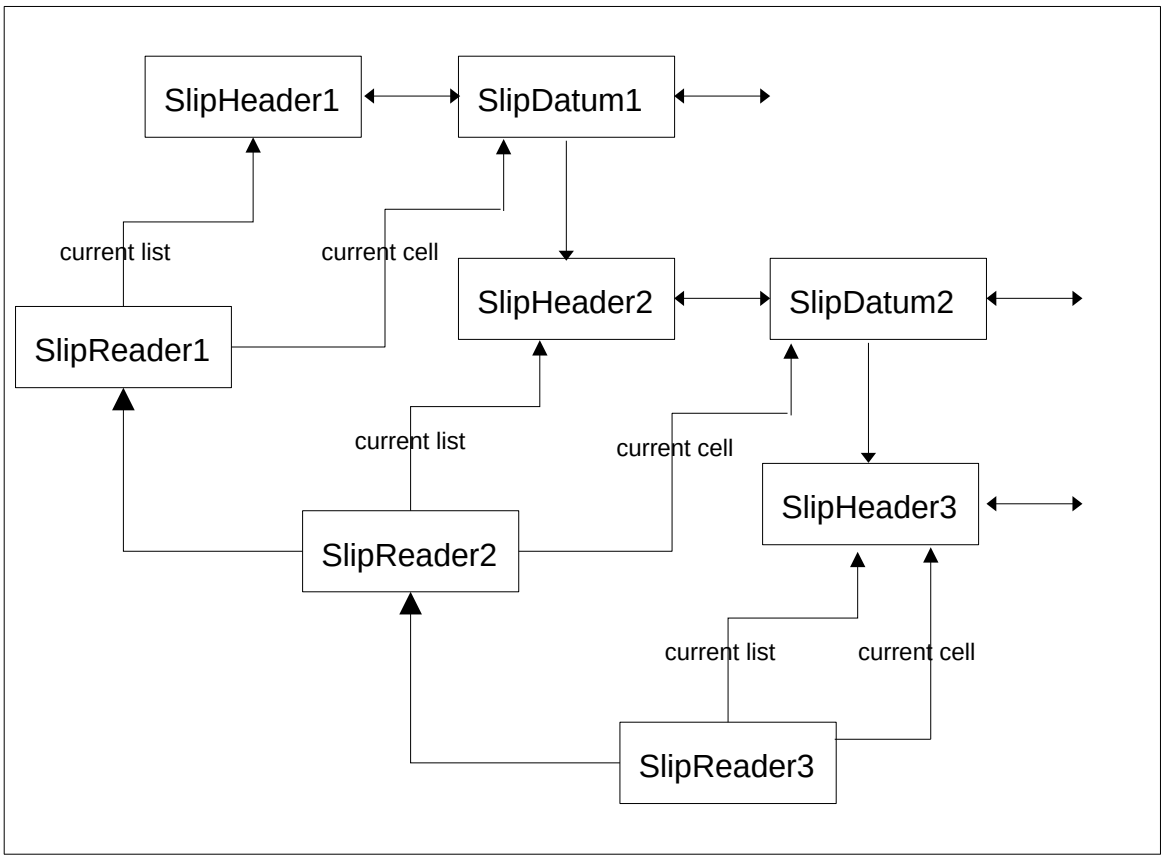


Figure 3.3 shows that an Iterator Partition has 4 components. The depth, reference to the previous Iterator Partition, a reference to the current list (SlipHeader), and a reference to the current object (SlipCell). The depth is the number of sublists traversed to the current sublist, with the initial list at a depth of zero and each additional traversed list adding one to the depth. The prev SlipReader is a reference to the previous Iterator Partition with the initial partition having a value of *nullptr*. The SlipHeader is a reference to the current list being traversed, and the SlipCell is a reference to the current cell within the list that the iterator is at.

The iterator is an analog for programming language function entry. When a function calls a function a stack frame is constructed. The frame contains a back pointer to the return address, and a list of arguments. The Iterator Partition contains a reference to the previous *instruction*, the previous Iterator Partition which contains a reference to the sublist causing a list descent, and the 'name' of the current list and current list cell being traversed.

During execution the *instruction pointer* is advanced to the next SlipCell in the current list, similarly to the instruction pointer advance druing program execution. When a sublist is traversed, a new Iteration Partition is created, and linked to its predecessor, before traversal in the sublist.



igure 3.4
SlipReader Usag

Figure 3.4 shows the iterator structure in action. As iteration proceeds the SlipCell reference changes. When a sluoblist is seen, and the application selects iteration to proceed to the referenced list, another 'frame' is constructed reference the current frame. Thereafter the created 'frame' becomes the operator frame in which iteration causes a modification of the SlipCell reference. At each entry to a descendent list, the depth is increased by 1.

In Figure 3.4:

1. The intial iterator depth is 0 and prev SlipReader is null The current list is SlipHeader1 and the current SlipCell is at SlipDatum1, a sublist.
2. The application chose to descend to the SlipHeader2 list, The current state of the iterator frame is that the depth is 1 and the prev SlipReader reference references SlipReader 1. The current list in the frame is SlipHeader2 and the current 'instruction' reference is to SlipDatum2, a sublist.
3. The application chose to descend to the SlipHeader3 list. The current state of the iterator frame is that the depth is 2 and the prev SlipReader reference references SlipReader 2. The current list in the frame is SlipHeader3 and the current 'instruction' reference is to SlipDatum3.

When the lists ascend to their parent, the frames are deleted and the return is to the sublist cell causing entry to the descendent list. The iteration function which causes entry to a descendent is now executed. After execution the current SlipCell reference will not be at the sublist, but at another SlipCell in the list.

4.0 Application Programming Interface (API)

The functional specifications fo the API are provided in this section. They are dividided into:

1. API Notes: identification and description of global API issues.
2. Common: Attributes and functions common to the SlipHeader, SlipDatum and SlipREader.
3. SlipHeader API: functions unique to the list header.
4. SlipDatum API: functions unique to the SlipDatum objects, data primitives and sublist objects.
5. SlipReader: functions and attributes of the SLIP iterator.

Both a SlipHeader (list header) and SlipDatum (data carrier) object must be from the AVSL. This is accomplished by ussing `new SlipHeader()` or `new SlipDatum(cell)`. Otherwise a list header will constrained to be empty, it can not have any list elements, and a SlipDatum object can not be inserted into a list, the contents will be copied, and can not be a sublist.

4.1 Common API

The SLIP iterator (SlipReader) contains a reference to both the list header (SlipHeader) and to the SlipDatum objects. The common functions apply to the current object that the iterator is referencing. They do not apply to the referenced SlipHeader object.

To execute a function in the runtime environent do:

1. ```
   SlipDatum data = new SlipDatum("something");
   data→function(args);
   ```
2. ```
   SlipHeader head = new SlipHeader();
   head→function(args);
   ```
3. ```
   SlipReader it = SlipIterator(new SlipHeader);
   it.function(args);            // or
   (it.datum()).function(args);
   ```

   Iterators default functions to refer to the current datum. If the application requires that the function is to be used on the list header (SlipHeader) and if the function is allowed to be used with a list header, then `(it.list()).function(args);` must be used.

All of the operations allow any of a `cell` data types, unless specifcally restricted. The interpretation of the argument is:

1. If the SlipDatum object contains `data` then the data is copied to a new SlipDatum cell.
   ```

2. If the SlipDatum object is a sublist then a new SlipDatum sublist is created referencing the same SlipHeader and the SlipHeader refcnt is incremented.
3. If the object is a list header (SlipHeader) then a sublist is created referencing the SlipHeader and the SlipHeader refcnt is incremented.

4.1.1 Common API Functions

These functions are active in the SlipDatum, SlipHeader and SlipReader objects. The particulars of usage differs depending on the object, and where so, it is indicated here and more fully described when the SLIP header object is directly addressed.

Common API Functions

string	cellStatus() ASCII output of Slip cell properties.

string	dump()
	Dump the content of the object and return it as a string. The dump provides detailed information on the object internals. The format of the dump depends on the cell type and contents. The dump format depends on the object. All pointers are in hex (0x) in the dump.
	
	All objects have an address (A:) and a pointer to an operations class (O:). All SlipDatum and SlipHeader objects have a pointer to the previous cell (L:) and the next cell (R:). SlipDatum objects have a value which is output, and SlipHeader objects have a reference count (refcnt) and application list mark (mrk), and Description List pointer (Desc).
	
	Iterators (SlipReader) have pointers to the previous iterator link (U), current header (H), current cell © and depth.
	
	The general format of the dump for each data type is:
	
	`[header ] A:0x O:0x L:0x R:0x refcnt: # Desc: 0x`
	`[data type] A:0x O:0x L:0x R:0x data`
	`[reader   ] A:0x O:0x U:0x H:0x C:0x depth: #`

void	dump(ostream&)

Dump the contents to the specified output stream. For example, `cell.dump(cout)` would output to standard output.

See dump().

classType getClassType()

Get the enumeration value representing the object. Each object has an enumerated value representing the type of the current object. The representation (`SlipDatum()`) is illustrative of the creation of objects of this type. SlipDef.h contains the enumeration.

enum ClassType	Description
`eUNDEFINED`	Unassigned AVSL object
`eBOOL`	`SlipDatum(bool)`
`eCHAR`	`SlipDatum(char)`
`eLONG`	`SlipDatum(long)`
`eUCHAR`	`SlipDatum(unsigned char)`
`eULONG`	`SlipDatum(unsigned long)`
`eDOUBLE`	`SlipDatum(double)`
`ePOINTER`	`TBS`
`eSTRING`	`SlipDatum(string&)`
`eHEADER`	`SlipHeader()`
`eREADER`	`SlipReader(new SlipHeader())`
`eSUBLIST`	`SlipDatum(new SlipHeader)`

string getName()

Common API Functions

Return the ASCII equivalent name of the object as a string. Each object has an ASCII equivalent name.

ASCII NAME	Description `SlipDatum()`
`UNDEFINED`	Unassigned AVSL object
`BOOL`	`SlipDatum(bool)`
`CHAR`	`SlipDatum(char)`
`LONG`	`SlipDatum(long)`
`UCHAR`	`SlipDatum(unsigned char)`
`ULONG`	`SlipDatum(unsigned long)`
`DOUBLE`	`SlipDatum(double)`
`STRING`	`SlipDatum(string&)`
`HEADER`	`SlipHeader()`
`READER`	`SlipReader(new SlipHeader())`
`SUBLIST`	`SlipDatum(new SlipHeader)`

SlipCell& insLeft(`cell`)

SlipCell& insright(`cell`)

Insert the argument (cell) to the left/right of the current object. The effect of this is that if the argument is:

Argument	Description `SlipDatum()`
`Bool true`	`new SlipDatum(true)` is inserted
`'a'`	`new SlipDatum('a')` is inserted.
`123`	`new SlipDatum(123)` is inserted.
`unsigned char 'c'`	`new SlipDatum(unsigned char 'c')` is inserted.
`unsigned int 123`	`new SlipDatum(unsigned 123)` is inserted.
`1.23`	`new SlipDatum(double 1.23)` is inserted.
`"string"`	`new SlipDatum("string")` is inserted.
`SlipHeader`	`New SlipDatum(SlipHeader)` is changed to a sublist entry and the referenced header refcnt is incremented.
`SlipReader`	The iterator datum cell (`it.datum()`)is copied and inserted. If the cell is a sublist then it is copied and the referenced SlipHeader refcnt is incremented.
`sublist`	`new SlipDatum(sublist)` is copied and the referenced SlipHeader refcnt is incremented.

string toString()

A string version of the object is created.

SlipHeader()	"HEADER".
SlipDatum(`data`)	Equivalent to cout << `data`.
SlipDatum(SlipHeader&)	Sublist is "SUBLIST"
SlipIterator()	Dump of current iterator cell.

string write()

Data fields are constructed as if output with quotes and 'u' appended as required. Sublist and list (SlipHeader) writes return the entire list, see 5.3 Output .

4.1.2 Common Predicates

Properties of SLIP cells can be interrogated. Either the current object has the property or it doesn't. There is no failure. `true` indicates that the current object has the property. False indicates the current object does not have the given property.

For iterators (SlipReader), the check applies to the current data object, that is: `it.is<property> ≡ (it.datum()).is<property>`.

Common Predicates

bool isAVSL()

Is the current object from the AVSL? The AVSL is the SLIP heap resource. If a SLIP object does not come from the AVSL, then its operations are limitied.

bool isBool()

Is the current object a SlipDatum cell containing boolean data.

bool isChar()

Is the current object a SlipDatum cell containing char data.

bool isData()

Is the current object a SlipDatum cell containing one of:

- bool
- char
- long
- double
- string
- unsigned char
- unsigned long

bool isDiscrete()

Is the current object a SlipDatum cell containing one of:

- bool
- char
- long
- unsigned char
- unsigned long

Common Predicates

bool isHeader()

Is the current object a list header (SlipHeader).

bool isName()

Is the current object a SlipDatum sublist. A sublist contains a reference to a list (SlipHeader).

bool isNumber()

Is the current object a SlipDatum cell containing one of:

bool
char
long
double
unsigned char
unsigned long

bool isReal()

Is the current object a SlipDatum cell containing double data.

bool isString()

Is the current object a SlipDatum cell containing string data.

4.1.3 System Common

SLIP systme internals and operating states can be interrogated. This supports debugging activities. Determine failure points in a list can be very difficult. The difficulty lies in the time difference between to instertion of an unexpected or illegal value into a list, and the subsequent time when the value is retrieved and is incorrect. In order to repair the error the point of false data entry is needed. But to detect the error at this time is difficult, hence, the error detection can occur many instructions after the occurrence. The system activities in support of SLIP activity are directly accessible. Although still difficult, it is possible to provide appropriate preconditions to allow some error detection capability.

System Common

SlipState getSlipState()

The current SLIP AVSL state is returned in a data structure. The SlipState structure is defined in SlipDef.h. It is:

```
struct SlipState {        // AVSL state
   ULONG  total;          // Total SLIP Cells
   ULONG  avail;          // Available SLIP Cells
   ULONG  alloc;          // Initial AVSL allocate
   ULONG  delta;          // Incremental AVSL allocation
   ULONG  counter;        // AVSL transactions
   ULONG  fragments;      // AVSL fragments

}
```

The AVSL is initially constructed using either default values for the allocation size (`alloc`) and increment size,(`delta`), or by the application use of `slipInit()` and updated through subsequent application of `slipInit()`. This establishes the number of SLIP cells in the creation of the AVSL (`alloc`) and determines the number of cells to append to the AVSL (`delta`) if the initial allocation is too small. Each allocation creates a fragment, with each fragment containing a portion of the total AVSL (`total`) cells allocated. AVSL data acquisitions (`new`) and recovery of AVSL cells (`delete`) are counted (`counter`) and cause a change in the number of AVSL cells available (`avail`).

void avslHistory(bool)

If the argument is set to `true` then each AVSL transaction, `new` or `delete`, causes a dump of the chosen SLIP cell to standard output. In the case of a `delete list`, the entire list is output.

bool isAvslHistory()

Checks the current status of the avslHistory flag.

void printAVSL(string)

Outputs AVSL header data and all SLIP cells that are not currently used, with diagnostic messages if the cells are illegally configured. The free cells have either never been used

or have been returned to the AVSL through a `delete`.

Each cell is verified for correctness with appropriate diagnostic messages output on integrity violations. An aggregate check is made to guarantee that the total cells counted agrees with the count in the AVSL state.

void printAVSLState(ostream&)

void printAVSLState(string&)

The AVSL state is output to standard output with a title or to a user supplied stream.

The AVSL state is:

cell size: the size of each SLIP cell
low address: Smallest address of any fragment rounded to cell size
high address:Largest address of any fragment rounded to cell size

void printClassSizes(ostream&)

The cell size of each SLIP class.

void printFragmentList(string)

Output each fragment to standard out. The AVSL consists of fragments acquired during processing, where a new fragment is acquired when the tatal available space in the AVSL is zero and more space is required. The size of each fragment is determined by the `alloc` size in the AVSL state.

The size of each fragment is determined by the last `slipInit()` executed, with a default provided if `slipInit()` is not executed.

void printMemory(string)

Dumps all AVSL cells to standard output. Each cell in each fragment is verified for correctness and output. Verification failure causes an output message to be generated. The output contains both the allocated and unalloceted cells.

void printSlipSizes(ostream&)

The size of all data structures used by SLIP are output to standard output.

4.2 SlipHeader

A list header (SlipHeader) contains references to a list and a Description List. The list is an unorganized list of data cells (SlipDadum) which can be organized into a stack or queue. The Description List is organized into key, value pairs, where they are both data cells.

The list can be searched and accessed linearly or structurally, see the SlipReader, where a linear access iterates through the top level list and the structural iteration can descend, depth first, into sublists.

A stack is like a stacked of plates where the last plate stacked is the first to be retrieved, last in, first out (LIFO). If the list is organized into a stack then the normal push/pop operatioins are applicable.

A quese is like a bank queue, where the first person in the queue is the next person to be served. First in, first out (FIFO). If the list is organized as a queue then the normal enqueue/dequeue operations are applicable.

Although it is possible to dynamically list access methods used, it is best to keep a consist view of the list. Either it is a list, stack or queue.

The Descriptor List is organized to allow searching by a key to retrieve a value, <key, value> pair. Operations in a Descriptor List involve pairs and not list elements.

A Descriptor List consists of one or more lists, each list organized as <key, value> pairs. Operations on the List will go through all the lists as required.

Note that a given object in the list or the Descriptor List is a SlipDatum cell, and a SlipDatum cell can be a sublist.

When structural access is required, all lists and sublists will be searched. Otherwise a linear search is made of the current list, no subist will be entered.

4.2.1 SlipHeader API

SlipHeader Constructor

SlipHeader() Constructs a SlipHeader. The initial field values are:
The list is empty.
top() == bot() == SlipHeader.
refcnt = 0.
mark = 0.
The refcnt is the number of sublists that the list is in. The range is [0 - 65,535].
The mark is an applilcation field available to mark a list. The range is [0 - 32,767]

SlipHeader Operators

object	op	object
SlipHeader	==	SlipHeader

The address of the objects are compared. If equal, true is returned.

sublist	==	SlipHeader

The sublist referenced SlipHeader address is compared, and if equal, true is returned. The objects iscan be reversed.

sublist	==	sublist

The sublist referenced SlipHeader address is compared, and if equal, true is returned.

SlipHeader API

SlipCell& bot()

Peek at the bottom object in the list. When the list is empty return a reference to the SlipHeader.

bool checkList()

Verify that the list is correct. The list will be structurally searched for incorrect objects. Diagnostic messaes will be output to standard out. `true` will be returned if there are no errors.

SlipCell& dequeue()

Remove and return the oldest object in the queue. When the queue is empty return a reference to the SlipHeader.

void dumpList()

Structurally dump each object in a list with diagnostic messages indicating any errors.

SlipHeader& enqueue(`cell`)

SlipHeader API

Add the `cell` to the oldest object in the queue. If the queue is empty, this become the first object.

uint16_t getMark()

Return the application mark, [0 – 32,767] from the list.

uint16_t getRefCount()

Return the number of sublists the list is contained in, [0 – 65,535].

bool isDList()

If the list contains a Descriptor List, return true.

bool isEmpty()

If the list is empty, return true.

bool isEqual(SlipHeader&)

Structural, topological and type equality are checked. In two lists if the type at a given position is not the same, the check fails. That is, a structural advance is made to both lists at the same step, the type values must be equal.

SlipHeader& flush()

Empty the list. Return all list cells to the AVSL.

SlipCell& pop()

Remove the newest `cell` from the stack. If the stack is empty, a reference to the SlipHeader is returned.

SlipHeader& push(SlipCell&)

Put the the SlipCell to the newest position on the stack. If the stack is empty this becomes the first entry.

unsigned short putMark(unsigned short)

Use the first 15-bits of the argument as the list mark.

SlipHeader API

SlipHeader&	read(string, int)
SlipHeader&	read(string, SlipOutputInterface&, int)
SlipHeader&	read(SlipInputInterface&, SlipOutputInterface&, int)
	Input an ASCII list, convert it into a valid list structure and return a SlipHeader reference to the constructed list. See Input/Output for more detail.
SlipHeader&	splitLeft(SlipCell&)
	Create a new list containing all cells from the list top() to the input cell, including the input cell, and remove the cells from the current list. The input argument can not be SlipHeader&.
SlipHeader&	splitRight(SlipCell&)
	Create a new list containing all cells from the input cell to the list bot(), including bot(), and remove the cells from the current list. The input argument can not be SlipHeader&.
SlipCell&	top()
	Return the object of the list top.
string	toString()
	Return "HEADER".
string	write(bool)
void	write(string&, bool = false)
void	write(SlipPrintInterface&, bool = false)
	Output the list. The list is formatted and output. A debug listing of internal operations is given if the boolean is true. See Input/Output.

4.2.2 SlipHeader Descriptor List

SlipHeader Descriptor List API

SlipHeader&	appendDList(SlipHeader&) Append a list containing <key, value> pairs to the Descriptor List of the current SlipHeader.
bool	contains(SlipCell&) Return true if the current Descriptor List <key, value> pairs contains the input value. Note, if there are mulitple key, value> pairs with the same value, only the first is considered.
bool	containsKey(SlipCell&) Return true if the current Descriptor List <key, value> pairs contains the input key. Note, if there are mulitple key, value> pairs with the same key, only the first is considered.
SlipHeader&	create_dList() Create a Descriptor List in the current SlipHeader. If one exists, ignore the operation. This creates an empty Descriptor List.
bool	deleteAttribute(SlipCell&) If the Descriptor List contains the key, delete the <key, value> pair. If there are multiple <key, value> pairs with the same key, only the first will be deleted.
SlipHeader&	delete_dList() Delete the Descriptor List. This creates an empty Descriptor List.
SlipHeader&	flush_dList() Empty the Descriptor List. Return all cells to the AVSL.
SlipCell*	get(SlipCell&) Return the value associated with the key in a <key, value> pair. If there are multiple keys then only the first is retrieved.

SlipHeader Descriptor List API

SlipHeader&	getDList() Return a reference to the current Descriptor List. If there is no Descriptor List then a fatal C++ error will occur.
vector<SlipCell*>	keys() Return a vector of SlipDatum references for all keys.
void	dumpDList() Same as dumpList except performed on the Descriptor List.
SlipHeader&	put(SlipCell&, SlipCell&) Insert a new <key, value> pair. • A Descriptor List will be created if none exists. • The key and value will be turned into a SlipDatum object as required.
unsigned	size_dList() The number of <key, value> pairs in a Description List.

4.3 SlipDatum

The two properties of SlipDatum cells are that they are list cells and that they are data carriers. As data carriers the SlipDatum cells are typed according to the data type carried by the cell. For example if the cell carries an int32_t then the SlipDatum *type* is integer.

Unlike strongly typed languages, such as C++, the types of SlipDatum cells are dynamically typed, hence the cells are heterogeneous and not homogeneous and compile time diagnostics can not be issued. When a cell type can not participate in an operation at runtime, then at runtime a diagnostic message is output.

SlipDatum cells can pariticpate in C++ operations in a manner identical to that of primitive elements of the language. They can be cast from one type to another, can participate in arithmetic, assignment, logical, and bitwise operations as appropriate to their types. Sublists have a more limited range of valid operations.

There are SLIP specific limitations on sublists and C++ limitations with strings and doubles which are listed in the API sections.

4.3.1 SlipDatum API

Except for a SlipSublis a SlipDatum and C++ atomic object can be on either side of a binary operator. That is for `x + y` either x, y, or both can be a SlipDatum object .

Sublists can not participate in unary, arithmetic, arithmetic assignment or bitwise operations and has a limited presence in logical and assignment operators. An `x = y` or `x == y` or `x != y` are supported, where x, y can be any SlipDatum or C++ atomic object.

SlipDatum Constructor

SlipDatum*	SlipDatum(SlipDatum&) Copy constructor. A new SlipDatum object is created with the type and value of the original. If the original is a sublist then the referernce SlipHeader object refcnt field is incremented.
SlipDatum*	SlipDatum(SlipHeader&) Create a sublist. A reference to the SlipHeader is used and the SlipHeader refcnt is incremented.
SlipDatum*	SlipDatum(`data`) The generated SlipDatum object has the type and value of the `data`. The `data` is copied.

SlipDatum API

SlipDatum&	moveLeft(SlipDatum&)
SlipDatum&	moveRight(SlipDatum&) Move the input cell to the right/left of the current cell. SlipHeader cells can not be moved. insLeft()/Right() is a better choice.
SlipDatum&	unlink() Remove the current cell from a list and return a reference.

4.3.2 SlipDatum Casting

SlipDatum objects which are sublists or strings can not be cast. All other data have available casts. The casts yield an atomic type in C++, not a SlipDatum object.

SlipDatum Casting Operators

bool	(bool) If the value of the SlipDatum object is 0 then the cast yields `false`, all other values are `true`.
char	(char) A SlipDatum discrete and double types can be cast to a char. Warnings for loss of precision is not given. Casts requiring greater than 8-bits are truncated.
int32_t	(int32_t) Loss of precision causes truncation not messages. This can occur for uint32_t and double types.
unsigned char	(unsigned char) All positive `discrete` types can be cast. Negative values will yield unspecified values. Casts requiring greater than 8-bits are truncated
uint32-t	(uint32-t) All positive `discrete` and double types can be cast. Negative values will yield unspecified values.
double	(double) All positive `discrete` and double types can be cast.

4.3.3 SlipDatum Unary Operators

In the below, var is a SlipDatum object, as in `SlipDatum var`.

Unary operators execute the C++ on the SlipDatum value. The C++ idioms are supported in the same way as the C++ types. Discrete types can participate. Non-discrtete types can not.

SlipDatum Unary Operators

bool	!var	logical NOT Does a C++ unary ! on the SlipDatum value. Note that a !double is double trouble.
uint32_t	~var	aritthmetic NOT (exclusive bitwise or) Does a C++ unary ~ on the SlipDatum value. Note that a ~double is double trouble.
SlipDatum&	+var	positive Has no effect on the SlipDatum value.
SlipDatum&	-var	change to negative value The SlipDatum value sign is reversed. This has an undefined effect on unsigned value types.
SlipDatum&	var++	post-increment Does a C++ post-increment on the SlipDatum value. Note that a double++ is not supported.
SlipDatum&	var--	post-decrement Does a C++ post-decrement on the SlipDatum value. Note that a double−− is not supported.
SlipDatum&	++var	pre-increment Does a C++ pre-increment on the SlipDatum value. Note that a ++double is not supported.
SlipDatum&	--var	pre-decrement Does a C++ pre-decrement on the SlipDatum value. Note that a double−− is not supported.

4.3.4 SlipDatum Assignment

An assignment operator causes the LHS of the assignment to be assigned the value given on the RHS. The RHS is not changed and can be either a SLIP cell or a C++ atomic object.

If the LHS is a string, then it is deleted before the assignment. If the LHS is a sublist, then the referenced SlipHeader cell is deleted, causing a decrement of the refcnt, and if 0, the deletion of the string.

If the RHS is a string then it is copied to the LHS. If the RHS is a sublist, then the referenced SlipHeader refcnt is incremented.

SlipDatum Assignment

SlipDatum&	=	`SlipDatum& = SlipHeader&` The SlipHeader reference is converted to a sublist and the SlipHeader refcnt is incremented.
SlipDatum&	=	`SlipDatum& = SlipDatum&` The type and value of the right hand side (RHS) of the equation is inserted into the left hand side (LHS) of the equation. If the original LHS was a sublist, then the referenced SlipHeader refnt is decremented, and if zero, the list is deleted.
SlipDatum&	=	`SlipDatum& = data` The type and value of the `data` is inserted into the LHS SlipDatum object. If the original LHS was a sublist, then the referenced SlipHeader refnt is decremented, and if zero, the list is deleted.
`data`	=	`Data = SlipDatum&` The right hand side (RHS) SlipDatum value is cast into the type of the left hand side (LHS) and substituted for the `data` value.

4.3.5 SlipDatum Arithmetic Assignment Operators

The Arithmetic Assignment Operators are only available for discrete types.

In all cases, `x op y` changes the value of x where x and/or y can be SlipDatum cells. The operations below are short hand notations for `x = x op y` or `x op y` as appropriate.

SlipDatum Arithmetic Assignment Operators

SlipDatum& `+=` `SlipDatum += SlipDatum& or SlipDatum += data`

Equivalent to `x = x + y`, where x and/or y can be SlipDatum cells.

SlipDatum& `-=` `SlipDatum -= SlipDatum& or SlipDatum -= data`

Equivalent to `x = x - y`, where x and/or y can be SlipDatum cells.

SlipDatum& `*=` `SlipDatum *= SlipDatum& or SlipDatum *= data`

Equivalent to `x = x * y`, where x and/or y can be SlipDatum cells.

SlipDatum& `/=` `SlipDatum /= SlipDatum& or SlipDatum /= data`

Equivalent to `x = x + y`, where x and/or y can be SlipDatum cells.

SlipDatum& `%=` `SlipDatum %= SlipDatum& or SlipDatum %= data`

Equivalent to `x = x % y`, where x and/or y can be SlipDatum cells.

SlipDatum& `<<=` `SlipDatum <<= SlipDatum& or SlipDatum <<= data`

Equivalent to `x = x << y`, where x and/or y can be SlipDatum cells. Bitwise *shift left*.

SlipDatum& `>>=` `SlipDatum >>= SlipDatum& or SlipDatum >>= data`

Equivalent to `x = x >> y`, where x and/or y can be SlipDatum cells. Bitwise *shift right*.

SlipDatum& `&=` `SlipDatum &= SlipDatum& or SlipDatum &= data`

Equivalent to `x = x & y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *and*.

SlipDatum& `|=` `SlipDatum |= SlipDatum& or SlipDatum |= data`

Equivalent to `x = x | y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *or*.

SlipDatum& `^=` `SlipDatum ^= SlipDatum& or SlipDatum ^= data`

SlipDatum Arithmetic Assignment Operators

Equivalent to `x = x ^ y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *exclusive or*

4.3.6 SlipDatum Binary Operators

Infix binary operators. They act the same as their assignment statement counterparts in Section 4.3.4.

SlipDatum Binary Operators

SlipDatum& + `SlipDatum + SlipDatum& or SlipDatum + data`

Equivalent to `x + y`, where x and/or y can be SlipDatum cells.

SlipDatum& - `SlipDatum - SlipDatum& or SlipDatum - data`

Equivalent to `x - y`, where x and/or y can be SlipDatum cells.

SlipDatum& * `SlipDatum * SlipDatum& or SlipDatum * data`

Equivalent to `x * y`, where x and/or y can be SlipDatum cells.

SlipDatum& / `SlipDatum /= SlipDatum& or SlipDatum / data`

Equivalent to `x / y`, where x and/or y can be SlipDatum cells.

SlipDatum& % `SlipDatum %= SlipDatum& or SlipDatum % data`

Equivalent to `x % y`, where x and/or y can be SlipDatum cells.

SlipDatum& << `SlipDatum << SlipDatum& or SlipDatum << data`

Equivalent to `x << y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *xhift left.*

SlipDatum& >> `SlipDatum >> SlipDatum& or SlipDatum >> data`

Equivalent to `x << y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *xhift right.*

SlipDatum& & `SlipDatum & SlipDatum& or SlipDatum & data`

SlipDatum Binary Operators

Equivalent to `x << y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *and.*

SlipDatum& | `SlipDatum | SlipDatum& or SlipDatum | data`

Equivalent to `x << y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *or.*

SlipDatum& ^ `SlipDatum ^ SlipDatum& or SlipDatum ^ data`

Equivalent to `x << y`, where x and/or y can be SlipDatum cells. Both x and y must be discrete types. Bitwise *exclusive or.*

4.3.7 SlipDatum Logical Operators

Sublist operations are limitied to == and `!=`, and can only compare two sublists. The references for the SlipHeaders are compared. Use of any other relational operator will yeild an error.

In `x op y`, either x, y, or both can be a SlipDatum object. Implicity casts are performed when the types of x or y are different. Note that this applies if x or y are C++ atomic types.

SlipDatum Relational Operators

bool != `SlipDatum != SlipDatum& or SlipDatum != data`

Equivalent to `x != y`, where x and/or y can be SlipDatum cells.

bool < `SlipDatum < SlipDatum& or SlipDatum < data`

Equivalent to `x < y`, where x and/or y can be SlipDatum cells.

bool <= `SlipDatum <= SlipDatum& or SlipDatum <= data`

Equivalent to `x <= y`, where x and/or y can be SlipDatum cells.

bool == `SlipDatum == SlipDatum& or SlipDatum == data`

Equivalent to `x == y`, where x and/or y can be SlipDatum cells.

bool >= `SlipDatum >= SlipDatum& or SlipDatum >= data`

SlipDatum Relational Operators

Equivalent to `x >= y`, where x and/or y can be SlipDatum cells.

bool > `SlipDatum > SlipDatum& or SlipDatum > data`

Equivalent to `x > y`, where x and/or y can be SlipDatum cells.

4.4 SlipReader

This is an iterator which allows advancement through a single list (linear) or advancement down sublists (structural). The advance functions allow advancement to the next (++) or previous (--) list elements of the current list, or searches to find sublists (names) or data (elements or SlipDatum).

A structural advance creates an `it` partition for each list descended into, and deleted the `it` partition for each list ascended from. This becomes the SLIP equivalent to a stack frame created druing function calls in a higher level language,

The iterator (`it`) allows access to the current list being traversed (`it.list()`) and the current item in the list being referenced (`it.datum()`), where the current data item is a SlipHeader or SlipDatum object. Unless `it.list()` is used to reference the list header, all Common API and SlipDatum API functions apply to the current data, `it.datum()`, see 4.4.2 SlipReader General API, for a list of exceptions.

Functional references to functionality, as in `it << it` can be used instead of `it.datum() << it.datum()` for a bitwise shift. The SlipHeader API is applied to the current list if the function used is unique to the SlipHeader. For example, `it.push(5)` applies to the current list but `it.insRight(5)` applies to the current data item. To apply `insRight(5)` to the current list, `(it.list()).insRight(5)` must be used, the extra parentheses forces C++ to evaluate `it.list()` before `insRight(5)`.

4.4.1 SlipReader Constructor API

Iterator Constructor

SlipReader*	SlipReader() Create an empty list and construct an iterator for it.
SlipReader*	SlipReader(SlipHeader&) Construct an interator for the input list.
SlipReader*	SlipReader(SlipReader&) Construct an iterator for the current list in the input iterator.
SlipReader*	SlipReader(SlipDatum&) Construct and iterator for the sublist header reference – the SlipDatum object must be a sublist.

4.4.2 SlipReader General API

When an API input argument is an iterator then the current iterator cell or current list reference is meant according to the function semantics.

Iterator Unary Operators

SlipCell&	*	Dereference iterator The iterator returns the `datum()` object to the user. The `datum()` object can be a SlipHeader or SlipDatum object. The common class for both is SlipCell. In order to ensure correct evaluation in complex processing surround `*it` with parenthesis as in (`*it`).
SlipReader&	var++	post-increment Advances to the next object in the current list. The next object becomes the current object (`datum()`).
SlipReader&	var--	post-decrement

Iterator Unary Operators

Advances to the previous object in the current list. The previous object becomes the current object (`datum()`).

SlipReader& ++var pre-increment

Advances to the next object in the current list. The next object becomes the current object (`datum()`).

SlipReader& --var pre-decrement

Advances to the previous object in the current list. The previous object becomes the current object (datum()).

Iterator General API

bool checkStack()

Verify the correctness of the iterator stack entries. During structured advances each time a sublist (a list contained within a lislt) is entered, a new iterator stack entry is created. This entry contains a reference to the previous entry, a reference to the current list (SlipHeader) being iterated over, and the current SlipCell. The correctness of the entries are verified starting at the current entry and proceeding to the starting (primary) list.

SlipCell& datum()

Return a reference to the current SlipCell.

SlipReader& deleteCell()

deleteDatum()

Delete the current cell. If the current cell is a SlipHeader, ignore the request. Advance the iterator to the previous cell is a SlipHeader, then do an upLevel() to the previous sublist. If there is no previous sublist, then the current cell is the SlipHeader. Advance functions will not stop (have the current cell) stop on any SlipHeader cell except the primary (topmost) SlipHeader. Allowing the previous cell become the current cell, and having this cell be a SlipHeader would break the contract. The only way that the current cell will ever be a SlipHeader will be if it is the topmost list.

SlipReader& flush()

Flush the current list of all SlipDatum cells and set the current cell to reference the SlipHeader. If this operation is performed while doing advances then this will break the requirement that the advance functions bypass all SlipHeader cells except the topmost list.

SlipHeader& list()

Return a reference to the current list being iterated over.

uint16_t listDepth()

Return the list depth defined as the number of sublists entered with the topmost list having a list depth of zero (0). As each sublist is entered, the list depth is increased. The maximum number of sublists entered is 65, 535.

Iterator General API

SlipCell& moveLeft(SlipReader&)

Move the input current cell to the left of the iterator current cell. Do nothing if the input current cell is a list (SlipHeader), use MoveListLeft() instead. The current cell reference in the input iterator is moved to the preceding cell (it--).

SlipCell& moveRight(SlipReader&)

Move the input current cell to the right of the iterator current cell. Do nothing if the input current cell is a list (SlipHeader), use MoveListRight() instead. The current cell reference in the input iterator is moved to the preceding cell (it--).

SlipReader& reset()

Reset the iterator to the start of the list. Make the current cell the current list.

SlipReader& resetTop()

Return to the topmost list. This operation sets the list depth to zero, the current list to the topmost list, and the current cell to the sublist causing descent into the first sublist. If the iterator is at the copmost list, `resetTop()` does nothing.

SlipReader& upLevel()

Ascend to the previous sublist. If at the topmost list, do nothing. The current cell after an `upLevel()` will be the sublist causing descent.

string toString()

Formats creates a string containing the list depth, current list and current cell reference for the current `it` partition.

string dump()

Dumps the contents of the current `it` partition.

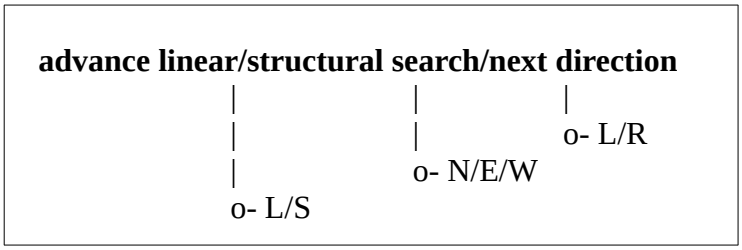
string dumpAll(const string&)

Dumps the contents of all active `it` partitions. The dump is prefixed by the optional title.

4.4.3 SlipReader Advance API

An iterator advance allows a regular advance over lists and sublists. The advance functions can advance to the next or previous items from the current position in a list (linear advances), can advance into and out of sublist (structural advances), can do a search across a single list (linear) or sublists (structural), and can go forwards to a next cell or backwards to a previous cell. The function names identify their purpose, direction and whether they are a search function or iteration.

The root name used for advances is "advance". To this root suffixes are applied in the following order: linear/structured, search/next, direction, as in:



The suffix definitions are:

- L: the first suffix character is Linear advance.
- S: structural advance. Advance to sublists.
- N: search for a sublist, and make it the current cell.
- E: search for a SlipDatum cell which is not a sublist, and make it the current cell.
- W: make the next cell the current cell.
- L: (last suffix character) move to the left (previous) direction.
- R: move to the right (next) direction.

Therefore we can interpret `advanceLWR()` as meaning advance linearly to the next cell to the right, equivalent to `it++`.

Structural advances cause the creation nad deletion of an Iteration Partition, see Section 3.3, each time a sublist is traversed. Except for the topmost list, when a sublist is 'entered' an Iteration Partition is created, when a sublist is 'exited' an Iteration Partition is deleted.

In search mode, advancexE/Nx, does a structural or linear search depending on the function. The search does not terminate at any list header except for the topmost list header. Additionally, during a structural advance, all sublist list headers are skipped over, that is, they are never a 'current cell'.

Linear advances advance in the current list. The current list may not be the topmost list, it can be any list that the iterator is positioned at. If a structural advance descends into the list of lists, then the current list is at some sublist to the topmost list.

Structural advances consider the topmost list header as the search terminator, unlike linear advances which will terminate at the current list list header. Starting at any sublist and doing a search, the search will terminate when the topmost list header is seen.

Iterator Linear Advances

SlipReader&	advanceLWR() advanceLWL() Advance to the next cell in the current list to the right (R) or left (L) of the current cell. This is equivalent to `it++` (`++it`) and `it—(--it`). The advance will advance to every list cell.
SlipReader&	advanceLEL() advanceLER() Advance to the next data cell in the current list. Ignore sublists. This is a search function which will continue searching for a data cell, ignoring sublists. It will stop at the topmost list header and report it as the current cell.
SlipReader&	advanceLNL() advanceLNR() Advance to the next sublist cell in the current list. Ignore data cells. This is a search function which will continue searching for a sublist cell, ignoring data cells. It will stop at the topmost list header and report it as the current cell.

Iterator Structural Advances

SlipReader& advanceSWR()
advanceSWL()

Advance to the next cell. If the next cell is a sublist then structurally descend into the sublist and create an Iterator Partition. If the next cell is a list header, the structurally ascend and delete the current Iterator Partition. The found cell or topmost list header becomes the current cell in the Iterator Partition.

SlipReader& advanceSEL()
advanceSER()

Advance to the next data cell in the current list. This is a search function. If the next cell is a sublist then structurally descend into the sublist and create an Iterator Partition and continue searching. If the next cell is a list header, then structurally ascend and delete the current Iterator Partition and continue searching. The found cell or topmost list header becomes the current cell in the Iterator Partition.

SlipReader& advanceSNL()
advanceSNR()

Advance to the next sublist cell in the current list and stop. This is a serch function. The sublist becomes the current cell in the Iterator Partition. If the list header of the current list is found then structurally ascend and delete the current Iterator Partition and continue searching. Terminate on a sublist cell or the topmost list header.

5.0 Input/Output

Input and Output are SlipHeader functions. You output a list, that is, a SlipHeader object. And you input to a list, that is, a SlipHeader object is created and populated from the input source. More on this in Sections 5.2 Input and, 5.3 Output .

The objective is to capture sufficient information to output a list in ASCII so that inputting the list will generate the same list. The Input and Output functions are inverses so that for any list L, `L = input(output(L))`. Inputting the ASCII representation of the output of a list should yield the same list.

This can be represented as

$I = O^{-1}$ where I stands for input, and O stands for output.

The I/O functions are meant to capture the complete environment of a list. A list contains a number of constituent part. There is a user defined 'mark' in each list header. The mark and its use is application defined. A list contains a linear list and a association list of <key, value> pairs where the key, the value, or both can be a list. And, each list can be contained within a list or within a number of lists. Data within SlipDatum cells have types, where the data type determines the type of output. All this information needs to be recorded on output and retrieved on input. Within this, a list which is contained in two or more lists should be created on input so that only one copy of the ilst exists, and on out so that multiple references exist. Capturing this requires a language, a means of specifying data types, multiple list references, the asscociation list, the list mark and so on.

5.1 I/O Language

The I/O language consists of a description section, which describes the allowable constructs that can be in a list or part of the description of the list topology, and allowable combinations of these primitive components into a valid list description. The primitive components can be part of any list description, but must be combined into a particular order in order for the description to be legal.

The description of the primitive components is defined by Regular Expressions. Regular expressions allow descpription of such things as what is a number, or, what is a string or a list start and stop. Combining the primitives is the function of the language syntax. The syntax says, for example, that a list terminator, '`)`', must follow a list initiator, '`(`', and defines the list'" `) a b c (`" as illegal because it violates these rules.

The 5.1.1.1 Regular Expressions defines the language which allows definition of tokens, and uses this language to define the tokens. 5.1.2 I/O Language Syntax defines the allowable combination of tokens for list output and list input.

5.1.1 Tokens

A formal description of the tokens available is given by first providing the meta-language in which the description is given. This formal description serves to unambiguously define the symbols used, and in what way they are used.

A token is something that is recognized as an object in the input/output description of a list. As a separate object, it stands for itself. The I/O Syntax determines where the tokens can exist. So, at the bottom of our language description are recognizable object, tokens, and the legitimacy of adjacent tokens are given in the syntax.

An informal description is given below. The informal description serves as a prelude to the formal description. So:

Informal Token Description

token	Category	Decription
BOF		Beginning of File. Absorbed during token analysis.
EOF		End of File. Absorbed during token analysis.
(		Left Parenthesis. Defines the start of a list.
)		Right Parenthesis. Defines to define the end of a list.
{	delimeter	Left Curly Brace. Defines the beginning of a list mark/sublist reference.
}		Right Curly Brace. Defines the end of a list mark/sublist reference.
<		Left Angle Bracket. Starts a descriptor list.
>		Right Angle Bracket. Ends a descriptor list.
space		Any of blank (‘ ‘), line feed (‘\n ‘), carriage return (‘\r ‘), tabs (‘\t’), bell (‘\b‘). form feed (‘\f‘)
[+|-]digits.[d][+|-]digits [+|-].digits[[d][+|-]digits]		A floating point number has an option sign, +|-, following by either digits and a period or a period and digits, following by an optional ‘d’ or ‘D’, an otional sign and mandatory digits.
true|false	bool	True and False are defined as boolean values.
character	char	A single alphabetic character is a ‘char’.
‘character’u		Quoted charaacter. A single slphabetic character followed by ‘u’ or

Informal Token Description

token	Category	Decription
		'U' is an unsigned 'char'.
a123_d	string	An alphabetic character followed by alphanumric characters is a string. A blank is not permitted. Underscores are allowed.
"a1*"		Quoted string. Any graphic character, including blanks.
[+|-]1 digits 'u'	integer	A decimal number has an optional sign followed by a digit > 0 followed by decimal digits. An optional 'u' or 'U' means the integer is unsigned.
0octal		An octal number begins with a '0' followed by octal digits.
0xhex		A hexadecimal number starts with 0x or 0X followed by hex digits.
\123	escape	Escaped integer, range [0 – 255]. Allowed in quoted character or string.
\0octal		Escaped octal value, [0 – 255]. Allowed in quoted character or string.
\0xhex		Escaped hesadecimal value, [0 – 255]. Allowed in quoted character or string.
\'		Escaped single quote. Allowed in quoted character or string.
\"		Escaped double quote. Allowed in quoted character or string.
\\		Escaped back slash. Allowed in quoted character or string.
\special		Any of \b, \f, \n, \r, \t. Allowed in quoted character and string.

5.1.1.1 Regular Expressions

Each allowed input or output token is defined using a regular expression. The regular expression is itself a language which allows tokens to be defined. It is defined below. Other resources should be used if more extent information is wanted. Examples of use are provided in the token definitions.

It is important to note that in regular expression blanks are important. A blank seen is recognized as a blank and not ignored. This is different than the descriptive mechanism used to define syntax. The syntax equations allow blanks, and where one blank is allowed, many can be used.

The descriptions follow a sequence of complexity.

1. The Special Characters, except for the Escape character, are suffixes to expressions and define the multiplicity of the expression.
2. The Grouping Elements allows subexpressions to be grouped, where a subexpression can be a single character.
3. Names of expressions are a shorthand for a set of characters, any one of which can be chosen.

Regular Expression Meta-Language
Special Characters

?	Match zero or one of the previous expression.
*	Match zero or many of the previous expression.
+	Match one or many of the previous expression.
\	Escape character. The character(s) following the escape character is special.

Regular Expression Meta-Language
Grouping Elements

[ ]	Set of values, where the sst is unordered and all members of the set are allowable. For example [abc123()&], meaning any member of the set.
-	Range, as in, 0-9 meaning all ASCII decimal digits from 0 to 9, the same as [:digits:].
|	Or, a choice between regular expressions, as in `a|b` meaning 'a' or 'b'.
( )	Grouping of regular expressions as in `(a|b)` meaning 'a' or 'b'.

Regular Expression Meta-Language
Names of Expressions

[:alnum:]	Alphanumeric characters: '[:alpha:]' and '[:digit:]'; in ASCII character encoding, This is the same as '[0-9A-Za-z]'.

Regular Expression Meta-Language
Names of Expressions

[:alpha:]	Alphabetic characters: '[:lower:]' and '[:upper:]'; in ASCII character encoding, this is the same as '[A-Za-z]'.
[:blank:]	Blank characters: space and tab.
[:digit:]	Digits: '0 1 2 3 4 5 6 7 8 9'.
[:graph:]	Graphical characters: '[:alnum:]' and '[:punct:]'.
[:punct:]	Punctuation characters; in ASCII character encoding, this is '! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ ] ^ _ ` { | } ~'.
[:space:]	Space characters: tab, newline, vertical tab, form feed, carriage return, and space.
[:xdigit:]	Hexadecimal digits: '0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f'.

5.1.1.2 Formal Token Description

Special tokens can only appear within a quoted character or quoted string. They can not appear outside of a quoted token. The values within the special tokens are evaluated and substituted. The resultant character or string will contain the substituted values. Since this is a character replacement, the values of any integer must be in the range of [0-255].

Special tokens

\b	Sound bell.
\f	Vertical form feed.
\n	Carriage return and line feed.
\r	Carriage return.
\t	Horizontal tab.
\1	Start of a decimal integer.
\0	Start of an octal integer.
\0x	Start of a hexadecimal integer.
\'	Single quote.
\"	Double quote.
\\	Back slash.

I/O Language Tokens

Regular Expression	Description
[[:alpha:]]	An input of type *signed char.* Example: a
'[[:graphic:]]'[u|U]?	A single, quoted charater followed by 'u' or 'U' is an *unsigned char* if 'u' or 'U' is present. This includes a blank and punctuation. Example: '<'
'\(b|n|f|r|t)'[u|U]?	A single character which is a special token. This is an *unsigned char* if 'u' or 'U' is present. Example: '\n'
'\0[0-7]*'(u|U)?	A binary octal number, optionally unsigned. Range [0 – 255]. Example: '\037'
'\[1-9][[:digit:]]*'(u|U)?	A binary decimal number, optionally unsigned. Range [0 – 255]. Example: '\255'
'\0(x|X)[[:xdigit:]]+'(u|U)?	A binary hexadecimal number, optionally unsigned. Range [0 – 255] (2 hex digits). Example: '\0x1F'
([[:alnum:]]|_)+	Two or more characters are a string. Blan-ks are not allowed and at least one non-decimal digit must be included. Example: George_III
"([[:graphic:]]|\(0[0-7]+|[1-9][[:digit:]]*|0(x|X)[[:xdigit:]])|\(b|n|f|r|t))*"	

I/O Language Tokens

Regular Expression	Description
	A quoted string is a sequence of quoted characters with quotes removed. Blanks are allowed, and any binary values are restricted to the range [0 – 255]. A string can be considered as a concatenated sequence of characters. An empty string is allowed (‟″). Note that a string is neither *signed* nor *unsigned*. Example "{Able \n\025 \t \63\t\0x1F \nBaker}"
(true|false)	Boolean. Both upper and lower case are allowed and can be mixed. Example: TRue
[+|-]?0[0-7]+	Octal number. The signed range is $[-2^{31} - +2^{31} – 1]$. Example: -012345 012345u
[+|-]?[1-9][[:digit:]]]+(u|U)?	Decimal number. The signed range is $[-2^{31} - +2^{31} – 1]$. The unsigned range is $[0 - 2^{32} – 1]$. Example: +12345 12345u
0(x|X)[[:xdigit:]]+	Hexadecimal number. **The range is $[0 - 2^{32} – 1]$.** **Example 0x12345**
[+|-]?[1-9][[:digit:]]]*.[[:digit:]]*[[e|E][+|-]?[[:digit:]]+] [+|-]?.[[:digit:]]].[[e|E][+|-]?[[:digit:]]+]	Floating point number. The signed or unsigned range is $1.7E^{-308}$ to $\pm 1.7E^{+308}$. Example +3.E-10 .554

5.1.2 I/O Language Syntax

The I/O Language supports linguistic definitions for SLIP internals. The internals define the full context of a list, what a list is, what a sublist is, and all constituent parts of a list. These parts, and the associated syntax, is defined below.

The blanks in the description are optional and are provided to give clarity to the description. All delimiters in the syntax are literal, that is, they must be present as displayed. The descriptions are illustrative. The complete syntax is given in the BNF.

List Syntax

Syntax Name	Syntax	Description
list mark	{ integer }	Application SlipHeader 'mark'.
sublist name	string: list	Sublist names must precede the list body and are not part of a list, that is, the name and its contents are not contained within a left/right parentheses. A list can be referenced in multple lists, each reference is a sublist reference. Defining a sublist by name allows reuse of the list by referencing the name.
sublist reference	{ string }	A reference to a named sublist.
Descriptor List	< sublist ... >	A Descriptor list is identified and a '<' and '>'. The association <key, value> pairs are contained within a sublist, and the Descriptor List can have multiple sublists, each with a list of <key, value> pairs. A Descriptor List must be the first object after a list opening left parenthesis. '('.
list	list mark (Descriptor List data ... sublist reference list ...)	The list mark and all parts listed, except for the '(' ,')' are optional. The description is recursive, a list can contain a list, and the list is unordered and the Descriptor List <key, value> pairs are unordered. There can be only one list defined in any I/O file.

Some examples may be helpful, in all cases only a single list is created, although there can be multiple named sublists:

Name	Example
Empty list:	`( )`
List w/data	`( 1 02 0x3 c 'd' string "string" )`
List w/Descriptor List	`( < (1 2 c 'd' string "string") > 1)`
List w/sublist reference	`sublist_1: (<(1 2 c 'd' string "string")> 19)` `( {sublist_1} )`

List w/everything	`dList: {0x15} ( 1 2 3 4 )`
	`sub_1: {03) { <{dList}> {dList} 3 )`
	`sub_2: ( )`
	`{0x20} ( <(5 6 7 8) {sub_1}> (9 10) {sub_2} 63 )`
	
	The defined reference lists must precede the list definition, can have list marks, can be empty, and can reference other defined sublist references. A Descriptor List is a list and can be both part of a Descriptor List and can be separately referenced as part of a list. A Descriptor List can have multiple lists, both referenced and not referenced, defined within its structure.

Empty list of lists	`( ( ( () ) ) )`
List of lists	`{0x1}( 1 {02} ( 2 {3} (3 ( 4 ) 6 ) 6 ) 7 )`

5.1.3 I/O Syntax

BNF places all of the I/O list elements into an organization.

The syntax consists of literals, denoted by quotes (''), tokens, such as integer or string, and BNF meta-syntax which consist os a LHS : RHS | RHS, where '":' acts as a separator between the LHS (Left Hand Side) and RHS (Right Hand Side) objects, and the vertical bar , '|' is an 'or' as in this 'or' that. Only one list can be defined.

```
I_O_List        : sublists list ;

sublists        : sublists sublist | sublist ;
```

```
sublist         : sublist_name ':' list ;

list            : list_mark '( descriptor_list list_objects ')' ;

descriptor_list: '<' list '>' ;

list_objects    : list_objects list_object
                | list_object ;

list_object     : list
                | '{' sublist_name '}'
                | data_list
                | ;

data_list       : data_list data
                | data ;

sublist_name    | string ;

list_mark       | '{' integer '}' ;

data            : bool
                | char
                | string
                | float
                | ;
```

5.2 Input

List readers are SlipHeader functions. The calling code is either `SlipHeader::read()`, or `obj.read()`; where obj is a SlipHeader object.

Each reader returns a reference to a single list, must be deleted (`delete &list)` when use of the list is no longer needed. The read function reads the entire list, any detected errors causes a return of an empty list with output going to the given output resource.

Each function contains a optional debug flag. The debug flag enable output of inormation gathered during input processing, with output going to the application designated output stream. The default is no debugging.

Debug Flag Values:

- debug = 0: [degault] Do not output debug information.
- debug = 1: Output debug information on syntax parsing.

- debug = 2: Output debug information on token lexing.
- debug = 3: Ouput debug information on both parsing and lexing.

SLIP Input Functions

Return value	Function Name
SlipHeader&	read(string) read(string, debug) This is the simplist input function. Read an ASCII version of the list from a string. All output goes to the standard stream, cout. Example: `SlipHeader& list = read("(1 (2) )");`
SlipHeader&	read(string, SlipOutputInterface&) read(string, SlipOutputInterface&, debug) Read an ASCII version of the list from a string, and output diagnostic and debug information to an application defined stream. The text of the SlipOutput Interface is given in Appendix C. Example: `SlipHeader& list = read("(1 (2) )", myOUt);`
SlipHeader&	read(SlipInputInterface&, SlipOutputInterface&) read(SlipInputInterface&, SlipOutputInterface&, debug) Read an ASCII version of the list from an application defined input stream, and output diagnostic and debug information to an application defined output stream. The text of the SlipOutput Interface is given in Appendix B. The text of the SlipOutput Interface is given in Appendix C. Example: `SlipHeader& list = read(myIn, myOUt);`

5.3 Output

Output a list (SlipHeader). The Description List and List associated with a SlipHeader object is formatted and output. The output conforms to the syntax and tokens identifies in 5.1.1.2 Formal Token Description and 5.1.2 I/O Language Syntax.

The given SlipHeader functions support this formatting and output to a string or to an application defined output stream. The functions support outputting debug information, processing flow and data

used an/or generated during output processing. The debug flag is a boolean with true indicating that debug information is required and false syspending such output. The default is no output

The SlipPrintInterface allows arbitrary output to a spplication defined stream, including cout. It's definition is provided in Appendix D, SlipPrintINterfaee.

Slip Output Functions

Return value	Function Name

string	write()
	write(bool)
	Return a string with a formatted ASCII output list.
	Example:
	`SlipHeader& list = *new SlipHeader();`
	`// construct a list`
	`cout << list.write();`

void	write(string&)
	write(string&, bool)
	Return a string with a formatted ASCII output list.
	Example:
	`SlipHeader& list = *new SlipHeader();`
	`string str =string();`
	`// construct a list`
	`list.write(str);`
	`cout << str;`

void	write(SlipPrintInterface&)
	write(SlipPrintInterface&, bool)
	Output an ASCII formatted list to the output interface, SlipPrintInterface.
	Example:
	`SlipHeader& list = *new SlipHeader();`
	`string str =string();`
	`SlipPrinterInterface& myOut = *new myInterface();`
	`// construct a list`
	`list.write(myOut);`

Appendix A
Application Program Interface (API)

SLIP Constructors

SlipDatum*	SlipDatum(SlipDatum&)	Copy constructor. Create a duplicate of the input.
SlipDatum*	SlipDatum(SlipHeader&)	Create a sublist.
SlipDatum*	SlipDatum(data)	Create a data SlipDatum object.
SlipHeader()	SlipHeader()	Create a header: refcnt=0, mark=0, Descriptor List = nullptr.
SlipReader*	SlipReader()	Create an iterator for an empty list.
SlipReader*	SlipReader(SlipHeader&)	Create an iterator for a list.
SlipReader*	SlipReader(SlipReader&)	Create an iterator from an iterator.
SlipReader*	SlipReader(SlipDatum&)	Create an iterator from a sublist.

Alphabetic API Listing

SlipReader&	advanceLEL()	Advance to next SlipDatum cell.
SlipReader&	advanceLER()	Advance to previous SlipDatum cell.
SlipReader&	advanceLNL()	Advance to next sublist.
SlipReader&	advanceLNR()	Advance to previous sublist.
SlipReader&	advanceLWL()	Advance to previous cell.
SlipReader&	advanceLWR()	Advance to next cell.
SlipReader&	advanceSEL()	Structure advance to next SlipDatum cell.
SlipReader&	advanceSER()	Structure advance to previous SlipDatum cell.
SlipReader&	advanceSNL()	Structure advance to next sublist.
SlipReader&	advanceSNR()	Structure advance to previous sublist.
SlipReader&	advanceSWL()	Structure advance to previous cell.
SlipReader&	advanceSWR()	Structure advance to next cell.
SlipHeader&	appendDList(SlipHeader&)	Append a Descriptor List.
void	advanceSER()	Set AVSL logging.
SlipCell&	bot()	Return list bottom cell.
string	cellStatus	ASCII output of the cell properties.
bool	checkList()	Verify list SLIP cell correctness.

Alphabetic API Listing

bool	checkStack()	Verify correctness iterator stack.
bool	contains(SlipCell&)	Does a Descriptor List(s) contain a value.
bool	containsKey(SlipCell&)	Does a Descriptor List(s) contain a key.
SlipHeader&	create_dList()	Create an empty Descriptor List.
SlipCell&	datum()	Return iterator current cell.
SlipHeader&	delete_dList()	Delete all Descriptor Lists.
bool	deleteAttribute(SlipCell&)	Delete a <key, value> pair from a Descriptor List.
SlipReader&	deleteDatum()	Delete iterator current cell.
SlipCell&	dequeue()	Remove and return oldest list cell.
string	dump()	Return the cell or iterator dump.
void	dump(ostream&)	Dump cell to the output stream.
string	dumpAll(const string&)	Dump all iterator list states with optional title.
void	dumpDList()	Dump <key value> pairs from a Descriptor List.
void	dumpList()	Dump each cell in list.
SlipHeader&	enqueue(cell)	Enqueue data to list (FIFO list).
SlipHeader&	flush_dList()	Flush all Descriptor List cells.
SlipHeader&	flush()	Empty the list or flush iterator current list.
SlipCell*	get(SlipCell&)	Get a Descriptor list value from <key, value> pair.
classType	getClassType()	Get the enumeration value representing the object
SlipHeader&	getDList()	Return a reference to a Descriptor List.
uint16_t	getMark()	Return value of application mark.
string	getName()	Return the ASCII name of the cell.
uint16_t	getRefCount()	Return list reference count.
SlipState	getSlipState()	SLIP AVSL state.
SlipCell&	insLeft(cell)	Insert to the left of the current cell.
SlipCell&	insright(cell)	Insert to the right of the current cell.
bool	isAVSL()	Is the cell from the AVSL?
bool	isAvslHistory()	Is AVSL logging on.

Alphabetic API Listing

bool	isBool()	Does the cell contain boolean data.
bool	isChar()	Does the cell contain char data.
bool	isData()	Is the cell a data cell.
bool	isDiscrete()	Does the cell have discrete data.
bool	isDList()	Does list have a Descriptor List.
bool	isEmpty()	Is the list empty.
bool	isEqual(SlipHeader&)	Are two list structurally equal.
bool	isHeader()	Is the cell a SlipHeader.
bool	isName()	Is the cell a sublist.
bool	isNumber()	Does the cell have a number.
bool	isReal()	Does the cell have a floating point number.
bool	isString()	Does the cell have a string.
vector<SlipCell*>	keys()	Return all keys from a Descriptor List.
SlipHeader&	list()	Iiterator current list.
uint16_t	listDepth()	Iterator current structured list depth.
SlipCell&	moveLeft(SlipDatum&)	Move iterator current cell to left of current cell.
SlipCell&	moveRight(SlipDatum&)	Move iterator current cell to right of current cell.
SlipCell&	pop()	Remove and return top stack entry.
void	printAVSL(string)	Log AVSL cells.
void	printAVSLState(ostream&)	Output to standard out SLIP AVSL state.
void	printAVSLState(string&)	
void	printClassSizes(ostream&)	Print SLIP cell sizes.
void	printFragmentList(string)	Output AVSL fragment headers.
void	printMemory(string)	Dumps used/free SLIP AVSL cells.
void	printSlipSizes(ostream&)	Ouput SLIP data structure sizes.
SlipHeader&	push(SlipCell&)	Push data to top of stack.
SlipHeader&	put(SlipCell&, SlipCell&)	Insert a <key, value> pair into a Descriptor List.
unsigned short	putMark(unsigned short)	Change the application list mark.

Alphabetic API Listing

SlipHeader&	read(SlipInputInterface&, SlipOutputInterface&, debug)	Read an input ASCII list.
SlipHeader&	read(string, int)	Read an input ASCII list.
SlipHeader&	read(string, SlipOutputInterface&, int)	Read an input ASCII list.
SlipReader&	reset()	Reset iterator to current list.
SlipReader&	resetTop()	Reset iterator to topmost list.
unsigned	size_dList()	Number of <key, value> pairs in a Descriptor List.
SlipDatum*	SlipDatum(data)	Construct a SlipDatum cell.
SlipDatum*	SlipDatum(SlipDatum&)	Construct a SlipDatum cell.
SlipDatum*	SlipDatum(SlipHeader&)	Construct a sublist cell.
SlipHeader*	SlipHeader()	Construct a list header.
SlipReader*	SlipReader()	Construct an iterator for an empty list.
SlipReader*	SlipReader(SlipDatum&)	Construct an iterator for the input sublist list.
SlipReader*	SlipReader(SlipHeader&)	Construct an iterator for the input list.
SlipReader*	SlipReader(SlipReader&)	Construct an iterator for the current list.
SlipHeader&	splitLeft(SlipCell&)	Create list with cells to the left and including the input.
SlipHeader&	splitRight(SlipCell&)	Create list with cells to the right and including the input.
SlipCell&	top()	Return the top stack entry.
string	toString()	String representing object.
SlipReader&	upLevel()	Return iterator to pervious list.
SlipDatum&	unlink()	
string	write()	Return an I/O version of the cell.
string	write(bool)	Return string with ASCII list, debug to cout.
void	write(SlipPrintInterface&)	Output a list to the output stream.
void	write(SlipPrintInterface&, bool)	Send ASCII list to stream, debug to cout.
void	write(SlipPrintInterface&)	Send ASCII list to stream.

Alphabetic API Listing

void	write(string&)	Output a list to the string.
void	write(string&, bool)	Output a list to the string, debug to cout.

Appendix B
SlipInputInterface

```
/**
   • @file    SlipInputInterface.h
   • @author  A. Schwarz
   • @date    December 19, 2024
   •
*/

using namespace std;

# include <string>

namespace slip {

/**
 * Support Lexer input operations.
 *
 * Separates input from mechanism for input allowing the input
    * stream to be an implementer choice.
```

```
   *
   * The only requuirement placed on the implementation, other than
   * the contract, is that an end of file is indicated by using the
   * EOF character.
   *
   * Implementation:
   * 1. Initial Condition: The line and column number is 0.
   * 2. Processing: The next character is 'raad' on get and the
   *    column number is increased by one.
   *       a. If the next character is a line feed, the line number
   *          is incremented by 1 and the column number is set
   *           to 1.
   * 3, No function can move beyond the End of File.
   *       a. getChar() will not move beyond the End of File.
   *       b.backup() will move to the character preceding the
   *           End of File.
   */
class SlipInputInterface {
protected:
int line    = 1;                    //!< line number
int col     = 0;                    //!< column number

public:
static const char END = '\0';       //!< End of File indicator

    /**
     * Get the next character in the input.
     *
     * The current character is read without advancing the line or
     * column number to the next character.
     *
     * @return current character
     */
     virtual char  getChar() = 0;

     /**
      * Look at the next character to be read without changing the
      * line or column. During a 'peek' the line and column numbers
      * are not changed. It is assumed that the character being
```

```
 * looked at is not read.
 *
 * @return The character following the last character read.
 */
 virtual char  peek() = 0;

/**
 * Rewind the input stream.
 *
 * Start over.
 */
 virtual void    rewind() = 0;

 /**
  * Return the current line number.
  *
  * A line number is an arbitrary definition. In general the
  * line number variable is incremented each time that a line
  * feed is read. The line numbers start at zero, that is, the
  * first 'line' is line number zero.
  *
  * @return line number
  */
  int     getLine() { return line; };

 /**
  * Return the character position of the current character
  * within a line.
  *
  * @return character position.
  */
  int     getCol() { return col; };

/**
 * Returns true if the last character in the stream has been
 * input.
 *
 * @return true if we are at the end-of-file.
 */
```

```
      virtual bool   isEOF() = 0;
   }; // class SlipInputInterface
}; // namespace slip
```

Appendix C
SlipOutputInterface

```
/**
   • @file    SlipOutputInterface.h
   • @author  A. Schwarz
   • @date    December 19, 2024
   •
*/

using namespace std;

# include <string>

namespace slip {
/**
 * Support parser/lexer output operations.
 *
 * The print function must be able to output a string. A string
 * contain a single line of output without a line ending (\n).
 *
```

```
 * @author A. Schwarz
 */

 /**
  * Output a text string generated by SlipWrite.
  */
class SlipOutputInterface
{
   public:
     /**
      * Output a text string generated by SlipWrite.
      */
      void virtual print(string str) = 0;

      /**
       * Flush the output stream
       */
      void virtual flush();
}; // interface SlipPrintInterface }; // namespace slip
```

Appendix D
SlipPrintInterface

```
/**
   • @file    SlipPrintInterface.h
   • @author  A. Schwarz
   • @date    December 19, 2024
   •
*/

using namespace std;

# include <string>

namespace slip {

   /**
    * Support parser/lexer output operations.
    *
    * The print function must be able to output a string. A string
    * contain a single line of output without a line ending (\n).
```

```
     *
     * @author A. Schwarz
     */

   /**
     * Output a text string generated by SlipWrite.
     */
   class SlipOutputInterface  {
   public:
      /**
       * Output a text string generated by SlipWrite.
       */
      void virtual print(string str) = 0;

      /**
       * Flush the output stream
       */
      void virtual flush();
   }; // interface SlipPrintInterface
}; // namespace slip
```