# THE PTOLEMY KERNEL

by

Joseph T. Buck

Memorandum No. UCB/ERL M93/8

19 January 1993

# THE PTOLEMY KERNEL

by

Joseph T. Buck

Memorandum No. UCB/ERL M93/8

19 January 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

# THE PTOLEMY KERNEL

by

Joseph T. Buck

Memorandum No. UCB/ERL M93/8

19 January 1993

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# The Ptolemy Kernel

A programmer's companion for Ptolemy 0.4
December 8, 1992

**by Joseph T. Buck**

This document describes the Ptolemy kernel. The principal audience for this document is programmers who seek to extend Ptolemy in major ways (e.g. write a new domain or a new parallel scheduler), or who seek a deeper understanding of how the kernel works. A detailed knowledge of C++ is assumed.

# Table of Contents

# 1 Basic concepts, classes, and facilities

This section describes some basic classes and low-level concepts that are used throughout Ptolemy. There are a number of iterator classes, all with the same interface. Several important non-class library functions are provided. A basic linked list class called SequentialList is heavily used. States (see Section 8.1 [class State], page 86) and Portholes (see Section 5.2 [class PortHole], page 57) can have *attributes*; these are particularly important in code generation. Finally, many of the significant classes in Ptolemy – functional blocks, portholes to implement connections, parameters – are derived from NamedObj, the basic object for implementing a named object that lives in a hierarchy.

## 1.1 The C++ Subset Used In Ptolemy

The Ptolemy system has grown up with the C++ language, so it does not use all the latest features in the newest compilers or every nook and cranny of Ellis and Stroustrup's Annotated Reference Manual, because of unimplemented features or lack of stability of implementation. Instead, we focused on stability. Accordingly, Ptolemy builds under the Gnu C++ compiler, version 2.2.2, with version 2.2 of the library `libg++`; it also build under Sun C++ 2.1, a port of the AT&T '`cfront`' compiler to the Sun.

This means, for one thing, that templates are not used. In addition, some features that do not work that well yet under g++, such as nested classes, are also avoided. Nested enumerations, however, are used in several places.

## 1.2 Iterators

Iterators are a very basic and widely used concept in Ptolemy, and are used repeatedly in Ptolemy programming in almost any situation where a composite object contains other objects. We have chosen to use a consistent interface for all iterator objects. The typical iterator class has the following basic interface (some iterators provide additional functions as well):

```
class MyIterator {
public:
    // constructor: argument is associated outer object
    MyIterator(OuterObject&);
    // next: return a pointer to the next object,
    // or a null pointer if no more
```

```
        InnerObject* next();
        // operator form: a synonym for next
        InnerObject* operator++() { return next(); }
        // reset the iterator to point to the first object
        void reset();
    }
```

A typical programming application for iterators might be something like

```
    // print the names of all objects in the container
    ListIter nextItem(myList);
    Item *itemP;
    while ((itemP = nextItem++) != 0)
        cout << itemP->name() << "\n";
```

It is, as a rule, not safe to modify most container classes in parallel with the use of an iterator, as the iterator may attempt to access an object that does not exist any more. However, the `reset` member function will always make the iterator safe to use even if the list has been modified (user-written iterators should preserve this property).

## 1.3 Non-class utility functions

The kernel provides several useful ordinary (non-class) functions, primarily for manipulating strings and pathnames. Some are defined in '`miscFuncs.h`', others in '`paths.h`'.

`char* savestring(const char*` *text*`);`

The `savestring` function creates a copy of the *text* argument with `new` and returns a pointer to it. It is the caller's responsibility to assure that the string is eventually deleted. The argument must not be a null pointer.

`const char* hashstring(const char*` *text*`);`

This function enters a copy of *text* into a hash table and returns a pointer to the entry. If two strings compare equal when passed to `strcmp`, then if both are passed to `hashstring`, the return values will be the same pointer.

`const char* expandPathName(const char*` *fileName*`);`

This function accepts a string and interprets it as a Unix pathname. If the string does not begin with a ~ or $ character, the string itself is returned. A leading "~/" is replaced by the user's home directory; a leading "~*user*/" is replaced by the home directory for *user*, unless there is no such user, in which case the original string is returned. Finally, a leading "$*env*" is replaced by the value of the environment variable *env*; if there is no such environment variable, the original string is returned.

If any substitutions are made, the return value is actually a pointer into a static buffer. This means that a second call to this function may write on top of a value returned by a previous call.

```
const char* pathSearch(const char* file,const char* path=0);
```

For this function, *path* is a series of Unix-style directory names, separated by colons. If no second argument is supplied or if the value is null, the value of the PATH environment variable is used instead. For each of the colon-separated directory strings, the function checks to see whether *dir/file* exists. If it finds a match, it returns a pointer to an internal buffer containing the full path of the match. If it does not find a match, it returns a null pointer.

```
int progNotFound(const char* program,const char* extra=0);
```

This function searches for *program* in the user's PATH using the `pathSearch` function. If a match is found, the function returns false (0). Otherwise it returns true (1) and also generates an error message with the `Error::abortRun` function. If the *extra* argument is given, it forms the second line of the error message.

## 1.4 Generic Data Structures

As Ptolemy does not use templates, our generic lists use the generic pointer technique, with

```
typedef void * Pointer;
```

The most commonly used generic data structure in Ptolemy is `SequentialList`. Other lists are, as a rule, privately inherited from this class, so that type safety can be preserved. It is possible to insert and retrieve items at either the head or the tail of the list.

## 1.5 Class SequentialList

This class implements a single linked list with a count of the number of elements. The constructor produces a properly initialized empty list, and the destructor deletes the links. However, the destructor does not delete the items that have been added to the list; this is not possible because it has only `void *` pointers and would not know how to delete the items.

There is an associated iterator class for SequentialList called ListIter.

### 1.5.1 SequentialList information functions

These functions return information about the SequentialList but do not modify it.

```
int size() const;
```

Return the size of the list.

```
Pointer head() const;
```

Return the first item from the list (0 if the list is empty). The list is not changed.

```
Pointer tail() const;
```

Return the last item from the list (0 if the list is empty). The list is not changed.

```
Pointer elem(int n) const;
```

Return the *n*th item on the list (0 if there are fewer than *n* items). Note that the time required is proportional to *n*.

```
int empty() const;
```

Return 1 if the list is empty, 0 if it is not.

```
int member(Pointer arg) const;
```

Return 1 if the list has a Pointer that is equal to *arg*, 0 if not.

### 1.5.2 other SequentialList functions

These functions modify the list they are applied to.

```
void prepend(Pointer p);
```

Add an item at the beginning of the list.

```
void append(Pointer p);
```

Add an item at the end of the list.

```
int remove(Pointer p);
```

Remove the pointer *p* from the list if it is present (the test is pointer equality). Return 1 if present, 0 if not.

```
Pointer getAndRemove();
```

Return and remove the head of the list. If the list is empty, return a null pointer (0).

```
Pointer getTailAndRemove();
```

Return and remove the last item on the list.

```
void initialize();
```

Remove all links from the list. This does not delete the items pointed to by the pointers that were on the list.

### 1.5.3 Class ListIter

ListIter is a standard iterator class for use with objects of class SequentialList. The constructor takes an argument of type

```
const SequentialList
```

and the `++` operator (or `next` function) returns a `Pointer`. Class ListIter is a friend of class SequentialList. In addition to the standard iterator functions `next` and `reset`, this class also provides a function

```
void reconnect(const SequentialList& newList);
```

that attaches the ListIter to a different SequentialList.

## 1.6 Doubly linked lists

Support for doubly linked lists is found in '`DoubleLink.h`'. The class DoubleLink implements a baseclass for nodes in the list, class DoubleLinkList implements the list itself, and class DoubleLinkIter forms an iterator.

We consider this class to have serious design flaws, so it may be reworked quite a bit in subsequent Ptolemy releases.

### 1.6.1 Class DoubleLink

Class DoubleLink is the base type for links in DoubleLinkList objects. There are two constructors:

```
DoubleLink(Pointer p, DoubleLink* next, DoubleLink* prev):
DoubleLink(Pointer p);
```

The first form initializes the *next* and *prev* pointers of the node as well as the contents. The second form sets these pointers to null.

```
Pointer content();
```

Returns the content pointer of the node.

```
virtual ~DoubleLink();
```

This is a do-nothing destructor, but making it virtual makes the right thing happen for derived classes.

```
void unlinkMe();
```

This method deletes the node from the list it is contained in, by connecting together the elements pointed to by the *prev* and *next* pointers.

The following data members are protected:

```
DoubleLink *next; // next node in the list
DoubleLink *prev; // previous node in the list
Pointer e;        // contents of this node
```

### 1.6.2 Class DoubleLinkList

```
DoubleLinkList();
DoubleLinkList(Pointer* e);
```

The first form creates an empty list. The second creates a one node list containing e; the node is on the heap.

```
virtual ~DoubleLinkList();
```

The destructor FIXME.

```
DoubleLink* createLink(Pointer p);
```

Create a new DoubleLink containing *p* on the heap. This is a design error: it has nothing to do with DoubleLinkList and should not be a member of this class.

```
void insertLink(DoubleLink *x);
void insert(Pointer e);
```

These functions insert at the beginning of the list. The first inserts a DoubleLink; the second creates a DoubleLink with `createLink` and inserts that.

```
void appendLink(DoubleLink *x);
void append(Pointer e);
```

These functions append at the end of the list. The first appends a DoubleLink; the second creates a DoubleLink with `createLink` and appends that.

```
void insertAhead(DoubleLink *y, DoubleLink *x);
void insertBehind(DoubleLink *y, DoubleLink *x);
```

The first one inserts y immediately ahead of x; the second inserts y immediately after x. Both these functions assume that x is in the list; disaster may result otherwise.

```
DoubleLink* unlink(DoubleLink *x);
```

Unlink the link x from the list and return a pointer to it. Make sure that x is in the list before calling this method, or disaster may result.

```
void removeLink(DoubleLink *x);
```

This is the same as `unlink`, except that x is deleted as well. The same cautions apply.

```
void remove(Pointer e);
```

The list is searched for a DoubleLink whose contents match e. If a match is found, the node is deleted. The search is made in the "forward" direction.

```
int find(Pointer e);
```

The list is searched for a DoubleLink whose contents match e. If a match is found, 1 (true) is returned; otherwise 0 (false) is returned.

```
virtual void initialize();
```

This function deletes all links of the list and makes the list empty.

```
void reset();
```

This function resets the fields of the DoubleLinkList object without doing anything to the nodes (why is this here?).

```
int size();
```

Return the number of elements in the list. Should be const.

```
DoubleLink *head();
DoubleLink *tail();
```

Return a pointer to the head, or to the tail, of the list. If the list is empty both will return a null pointer.

```
DoubleLink *getHeadLink();
Pointer takeFromFront();
```

The former function gets and removes the head link from the list, returning a pointer to it. The second function calls the first to obtain the head node; it gets the node, saves the contents, deletes the node, and returns the contents. If the list is empty, both functions return a null pointer.

```
DoubleLink *getTailLink();
Pointer takeFromBack();
```

These functions are identical to the previous pair except that they remove the last node rather than the first.

The following two data members are protected:

```
DoubleLink *myHead;
DoubleLink *myTail;
```

### 1.6.3 Class DoubleLinkIter

DoubleLinkIter is an iterator for DoubleLinkList. It is only capable of moving "forward" through the list (following the "next" links, not the "prev" links).

Its `next` operator returns the Pointer values contained within the nodes; it is also possible to use the non-standard `nextLink` function to return successive DoubleLink pointers.

## 1.7 Other generic container classes

The file DataStruct.h defines two other generic container classes that are privately derived from SequentialList: Queue and Stack.

Class Queue may be used to implement a FIFO or a LIFO queue, or a mixture.

Class Stack implements a stack.

### 1.7.1 Class Queue

The constructor for class Queue builds an empty queue. The following four functions move pointers into or out of the queue:

```
void putTail(Pointer p);
void putHead(Pointer p);
Pointer getHead();
Pointer getTail();
```

put is a synonym for putTail, and get is a synonym for getHead. All these functions are implemented on top of the (hidden) SequentialList functions. The SequentialList functions size and initialize are re-exported (that is, are accessible as public member functions of class Stack).

### 1.7.2 Class Stack

The constructor for class Stack builds an empty stack. The following functions move pointers onto or off of the stack:

```
void pushTop(Pointer p);
Pointer popTop();
pushBottom(Pointer p);
```

pushTop and popTop are the functions traditionally associated with a stack; pushBottom adds an item at the bottom, which is non-traditional. The following non-destructive function also exists:

```
Pointer accessTop() const;
```

It accesses but does not remove the element from the top of the stack.

All these functions are implemented on top of the (hidden) SequentialList functions. The SequentialList functions `size` and `initialize` are re-exported.

## 1.8 Class NamedObj

NamedObj is the baseclass for most of the common Ptolemy objects. A NamedObj is, simply put, a named object; in addition to a name, a NamedObj has a pointer to a parent object, which is always a Block (a type of NamedObj). This pointer can be null. A NamedObj also has a descriptor.

Warning! NamedObj assumes that the name and descriptor "live" as long as the NamedObj does. They are not deleted by the destructor, so that they can be compile-time strings.

Important derived types of NamedObj include Block (see Section 2.1 [class Block], page 17), GenericPort (see Section 5.1 [class GenericPort], page 54), State (see Section 8.1 [class State], page 86), and Geodesic (see Section 5.6 [class Geodesic], page 68).

### 1.8.1 NamedObj constructors and destructors

All constructors and destructors are public.

NamedObj has a default constructor, which sets the name and descriptor to empty strings and the parent pointer to null, and a three-argument constructor:

`NamedObj(const char*` *name*`,Block*` *parent*`,const char*` *descriptor*`)`

NamedObj's destructor is virtual and does nothing.

### 1.8.2 NamedObj public members

```
virtual const char* className() const;
```

className returns the name of the class. It should have a new implementation supplied for every derived class (except for abstract classes, where this is not necessary).

```
const char* name() const;
```

name returns the local portion of the name of the class.

```
const char* descriptor() const;
```

descriptor returns the descriptor.

```
Block* parent() const;
```

parent returns a pointer to the parent block.

```
virtual StringList fullName() const;
```

`fullName` returns the full name of the object. This has no relation to the class name; it specifies the specific instance's place in the universe-galaxy-star hierarchy. The default implementation returns names that might look like

```
universe.galaxy.star.port
```

for a porthole; the output is the fullName of the parent, plus a period, plus the fullName of the NamedObj it is called on.

`void setNameParent (const char*` *my_name*`,Block*` *my_parent*`)`

This method changes the name and parent pointer of the object.

```
virtual void initialize() = 0;
```

initialize is a pure virtual method. Its function is to initialize the object to prepare for system execution.

`virtual StringList print (int` *verbose*`) const;`

print returns a verbose description of the object. If verbose is 0, a somewhat more compact form is printed than if verbose is 1.

```
virtual int isA(const char* cname) const;
```

The isA method should be redefined for all classed derived from NamedObj. Its function is to return TRUE if the argument is the name either of the class or of one of the baseclasses. To make this easy to implement, a macro `ISA_FUNC` is provided; for example, in the file `Block.cc` we see the line

```
ISA_FUNC(Block,NamedObj);
```

since NamedObj is the base class from which Block is derived. This macro creates the function definition

```
int Block::isA(const char* cname) const {
        if (strcmp(cname,"Block") == 0) return TRUE;
        else return NamedObj::isA(cname);
}
```

Methods `isA` and `className` are overriden in derived classes; the redefinitions will not be described for each individual class.

### 1.8.3 NamedObj protected members

```
void setDescriptor(const char* desc);
```

The descriptor is set to *desc*. The string pointed to by *desc* must live as long as the NamedObj does.

## 1.9 Class NamedObjList

Class NamedObjList is simply a list of objects of class NamedObj. It is privately inherited from class SequentialList (see Section 1.5 [class SequentialList], page 4), and, as a rule, other classes privately inherit from it. It supports only a subset of the operations provided by SequentialList; in particular, objects are added only to the end of the list. It provides extra operations, like searching for an object by name and deleting objects.

This object enforces the rule that only const pointers to members can be obtained if the list is itself const; hence, two versions of some functions are provided.

### 1.9.1 NamedObjList information functions

The `size` and `initialize` functions of SequentialList are re-exported. Note that `initialize` removes only the links to the objects and does not delete the objects.

Here's what's new:

```
const NamedObj* objWithName(const char* name) const;
NamedObj* objWithName(const char* name);
```

Find the first NamedObj on the list whose name is equal to *name*, and return a pointer to it. Return 0 if it is not found.

```
NamedObj* head();
const NamedObj* head() const;
```

Return a pointer to the first object on the list (0 if none). There are two forms, one of which can be applied to const NamedObjList objects.

### 1.9.2 other NamedObjList functions

```
void put(NamedObj& obj)
```

Add a pointer to *obj* to the list, at the end.

```
void initElements();
```

Apply the `initialize` method to each NamedObj on the list.

```
int remove(NamedObj* obj);
```

Remove *obj* from the list, if present (this does not delete *obj*). Return 1 if it was present, 0 if not.

```
void deleteAll();
```

Delete all elements from the list, and reset it to be an empty list. WARNING: this assumes that the members of the list are on the heap (allocated by `new`, so that deleting them is valid)!

### 1.9.3 NamedObjList iterators

There are two different iterators associated with NamedObjList; class NamedObjListIter and class CNamedObjListIter. The latter may be applied to const NamedObjList objects and returns const NamedObj pointers; the former requires non-const NamedObjList objects and returns non-const NamedObj pointers. They obey the standard iterator interface and are privately derived from class ListIter.

## 1.10 Attributes

Attributes represent logical properties that an object may or may not have. A parameter may or may not be settable by the user; an assembly-language buffer may be allocated in ROM or RAM, fast memory or slow memory, etc.

The set of attributes of an object is stored in an entity called a bitWord. At present, a bitWord is represented as an unsigned long, which restricts the number of distinct attributes to 32; this may be changed in future releases.

An Attribute object represents a request to turn certain attributes of an object off, and to turn other attributes on. As a rule, constants of class Attribute are used to represent attributes, and users have no need to know whether a given property is represented by a true or false bit in the bitWord.

Although we would prefer to have a constructor for Attribute objects of the form

```
Attribute(bitWord bitsOn, bitWord bitsOff);
```

it has turned out that doing so presents severe problems with order of construction, since a number of global Attribute objects are used and there is no simple, portable way of guaranteeing that these objects are constructed before any use. As a result, the `bitsOn` and `bitsOff` members are public, but we forbid use of that fact except in one place: constant Attribute objects can be initialized by the C "aggregate form", as in the following example:

```
extern const Attribute P_HIDDEN = {PB_HIDDEN, 0};
```

The first word specified is the `bitsOn` field, and the second word specified is the `bitsOff` field. Other than to initialize objects, we pretend that these data members are private.

### 1.10.1 Attribute member functions

```
Attribute& operator |= (const Attribute& arg);
Attribute& operator &= (const Attribute& arg);
```

These operations combine attributes, by applying the |= and &= operators to the bitsOn and bitsOff fields. The first operation, as attributes are commonly used, represents a requirement that two sets of attributes be met, so it has been argued that it really should be the "and" operation. However, the current scheme has the virtue of consistency.

```
bitWord eval(bitWord defaultVal) const;
```

Evaluate an attribute given a default value. Essentially, bits corresponding to bitsOn are turned on, and then bits corresponding to bitsOff are turned off.

```
bitWord clearAttribs(bitWord defaultVal) const;
```

This method essentially applies the attribute backwards, reversing the roles of bitsOn and bitsOff in `eval`.

```
bitWord on() const;
bitWord off() const;
```

Retrieve the bitsOn and bitsOff values, respectively.

Inline definitions of operators & and | are also defined to implement nondestructive forms of the &= and |= operations.

# 2 Block and related classes

This section describes Block, the basic functional block class, and those objects derived from it. It is Blocks more than anything else that a user of Ptolemy deals with. Actors as well as collections of actors are Blocks.

Although the Target class is derived from class Block, it is documented elsewhere, as it falls under control of execution (see Section 3.1 [class Target], page 35).

## 2.1 Class Block

Block is the basic object for representing an actor in Ptolemy. It is derived from NamedObj (see Section 1.8 [class NamedObj], page 11).

Important derived types of Block are Star (see Section 2.2 [class Star], page 22), representing an atomic actor; Galaxy (see Section 2.3 [class Galaxy], page 23), representing a collection of actors that can be thought of as one actor, and Universe (see Section 2.7 [class Universe], page 32), representing an entire runnable system.

A Block has portholes (connections to other blocks — see Section 5.2 [class PortHole], page 57), states (parameters and internal states — see Section 8.1 [class State], page 86), and multiportholes (organized collections of portholes — see Section 5.3 [class MultiPortHole], page 63).

While the exact data structure used to represent each is a secret of class Block, it is visible that there is an order to each list, in that iterators return the contained states, portholes, and multiportholes in this order. Iterators (see Section 1.2 [Iterators], page 1) are a set of helper classes that step through the states, portholes, or multiportholes that belong to the Block, see the menu entry.

Furthermore, Blocks can be cloned, an operation that produces a duplicate block. There are two cloning functions: `makeNew`, which resembles making a new block of the same class, and `clone`, which makes a more exact duplicate (with the same values for states, for example). This feature is used by the KnownBlock class (see Section 9.1 [class KnownBlock], page 94) to create blocks on demand.

### 2.1.1 Block constructors and destructors

Block has a default constructor, which sets the name and descriptor to empty strings and the parent pointer to null, and a three-argument constructor:

```
Block(const char* name,Block* parent,const char* descriptor);
```

Block's destructor is virtual and does nothing, except for the standard action of destroying the Block's data members.

In addition, Block possesses two types of "virtual constructors", the public member functions `makeNew` and `clone`.

### 2.1.2 Block public "information" members

```
int numberPorts() const;
int numberMPHs() const;
int numberStates() const;
```

The above functions return the number of ports, the number of multiports, or the number of states in the Block.

```
virtual int isItAtomic() const;
virtual int isItWormhole() const;
```

These functions return TRUE or FALSE, based on whether the Block is atomic or not, or a wormhole or not. The base implementations return TRUE for isItAtomic, FALSE for isItWormhole.

```
virtual StringList print(int verbose) const;
```

Overrides `NamedObj::print`. This function gives a basic printout of the information in the block.

```
GenericPort* genPortWithName(const char* name);
PortHole* portWithName(const char* name);
MultiPortHole* multiPortWithName(const char* name);
virtual State *stateWithName(const char* name);
```

These functions search the appropriate list and return a pointer to the contained object with the matching name. `genPortWithName` searches both the multiport and the regular port lists (multiports first). If a match is found, it returns a pointer to the matching object as a `GenericPort` pointer.

```
int multiPortNames (const char** names, const char** types,
                    int* io, int nMax) const;
```

Get a list of multiport names.

```
StringList printPorts(const char* type, int verbose) const;
```

Print portholes as part of the info-printing method.

```
virtual Scheduler* scheduler() const;
```

Return the controlling scheduler (see Section 3.2 [class Scheduler], page 40) for this block. The default implementation simply recursively calls the `scheduler()` function on the parent, or returns 0 if there is no parent. The intent is that eventually a block with a scheduler will be reached (the top-level universe has a scheduler, and so do wormholes).

```
virtual Star& asStar();
virtual const Star& asStar() const;
```

Return reference to me as a Star, if I am one. Warning: it is a fatal error (the entire program will halt with an error message) if this method is invoked on a Galaxy! Check with `isItAtomic` before calling it.

```
virtual Galaxy& asGalaxy();
virtual const Galaxy& asGalaxy() const;
```

Return reference to me as a Galaxy, if I am one. Warning: it is a fatal error (the entire program will halt) if this method is invoked on a Star! Check with `isItAtomic` before calling it.

```
virtual const char* domain() const;
```

Return my domain (e.g. SDF, DE, etc.)

### 2.1.3 other Block public members

```
virtual void initialize();
```

overrides NamedObj::initialize. Block::initialize initializes the portholes and states belonging to the block, and calls `setup()`, which is intended to be the "user-supplied" initialization function.

```
virtual int run();
```

This function is intended to "run" the block. The default implementation does nothing.

```
virtual void wrapup();
```

This function is intended to be run after the completion of execution of a universe, and provides a place for wrapup code. The default does nothing.

```
virtual Block& setBlock(const char* name,Block* parent=0);
```

Set the name and parent of a block.

```
virtual Block* makeNew() const
```

This is a very important function. It is intended to be overloaded in such a way that calling it produces a newly constructed object of the same type. The default implementation causes an error. Every derived type should redefine this function. Here is an example implementation of an override for this function:

```
Block* MyClass::makeNew() const { return new MyClass;}
```

```
virtual Block* clone() const
```

The distinction between `clone` and `makeNew` is that the former does some extra copying. The default implementation calls `makeNew` and then `copyStates`; it may be overridden to copy more information. The intent is that `clone` should produce an identical object.

```
void addPort(PortHole& port)
void addPort(MultiPortHole& port)
```

Add a porthole, or a multiporthole, to the block's list of known ports or multiports.

```
int removePort(PortHole& port)
```

Remove *port* from the Block's port list, if it is present. 1 is returned if *port* was present and 0 is returned if it was not. Note that *port* is not deleted. The destructor for class PortHole calls this function on its parent block.

```
void addState(State& s);
```

Add the state *s* to the Block's state list.

```
virtual void initState();
```

Initialize the States contained in the Block's state list.

```
StringList printStates(const char* type,int verbose) const;
```

Return a printed representation of the states in the Block. This function is used as part of the Block's `print` method.

```
int setState(const char* stateName, const char* expression);
```

Search for a state in the block named *stateName*. If not found, return 0. If found, set its initial value to *expression* and return 1.

### 2.1.4 Block protected members

```
virtual void setup();
```

User-specified additional initialization. By default, it does nothing. It is called by Block::initialize (and should also be called if initialize is redefined).

```
Block* copyStates(const Block& src);
```

method for copying states during cloning. It is designed for use by clone methods, and it assumes that the src argument has the same state list as the `this` object.

### 2.1.5 Block iterator classes

There are three types of iterators that may be used on Blocks: BlockPortIter, BlockStateIter, and BlockMPHIter. Each takes one argument for its constructor, a reference to Block. They step through the portholes, states, or multiportholes, of the Block, respectively, using the standard iterator interface.

There are also variant versions with a "C" prefix (CBlockPortIter, etc) defined in the file '`ConstIters.h`' that take a reference to a const Block and return a const pointer.

## 2.2 Class Star

Class Star represents the basic executable atomic version of Block. It is derived from Block (see Section 2.1 [class Block], page 17).

Stars have an associated Target (see Section 3.1 [class Target], page 35), an index value, and an indication of whether or not there is internal state.

The default constructor sets the target pointer to null, sets the internal state flag to TRUE, and sets the index value to -1.

### 2.2.1 Star public members

```
int run();
```

Execute the Star. This method also interfaces to the SimControl class to provide for control over simulations. All derived classes that override this method must invoke `Star::run`.

```
StringList print (int verbose = 0) const;
```

Print out info on the star.

```
Star& asStar();
const Star& asStar() const;
```

These simply return a reference to `this`, overriding `Block::asStar`.

```
int index() const;
```

Return the index value for this star. Index values are a feature that assists with certain schedulers; the idea is to assign a numeric index to each star at any level of a particular Universe or Galaxy.

```
virtual void setTarget(Target* t);
```

Set the target associated with this star.

```
void noInternalState();
```

Declare that this star has no internal state (This function may change to protected in future Ptolemy releases).

```
int hasInternalState();
```

Return TRUE if this star has internal state, false if it doesn't. Useful in parallel scheduling.

### 2.2.2 Star protected members

```
virtual void go();
```

This is a method that is intended to be overriden to provide the principal action of executing this block. It is protected and is intended to be called from the `run()` member function. The separation is so that actions common to a domain can be provided in the run function, leaving the writer of a functional block to only implement `go()`.

```
Target* targetPtr;
```

This is a public data member, set by the setTarget public member function.

## 2.3 Class Galaxy

A Galaxy is a type of Block (see Section 2.1 [class Block], page 17) that has an internal hierarchical structure. In particular, it contains other Blocks (some of which may also be galaxies). It is possible to access only the top-level blocks or to flatten the hierarchy and step through all the blocks, by means of the various iterator classes associated with Galaxy.

While we generally define a different derived type of Star for each domain, the same kinds of Galaxy (and derived classes such as InterpGalaxy — see Section 2.5 [class InterpGalaxy], page 27) are used in each domain. Accordingly, a Galaxy has a data member containing its associated domain (which is set to null by the constructor).

PortHoles belonging to a Galaxy are, as a rule, aliased so that they refer to PortHoles of an interior Block, although this is not a requirement.

## 2.3.1 Galaxy public members

`void initialize();`

System initialize method. Derived Galaxies should not redefine initialize; they should write a `setup()` method to do any class-specific startup.

`void wrapup();`

System wrapup method. Recursively calls wrapup in subsystems

**void addBlock(Block&** *b*,**const char*** *bname***);**

Add block to the galaxy and set its name.

**int removeBlock(Block&** *b***);**

Remove the block *b* from the galaxy's list of blocks, if it is in the list. The block is not deleted. If the block was present, 1 is returned; otherwise 0 is returned.

`virtual void initState();`

Initialize states.

`int numberBlocks() const;`

Return the number of blocks in the galaxy.

**StringList print(int** *verbose***) const;**

Print a description of the galaxy.

```
int isItAtomic () const;
```

Returns FALSE (galaxies are not atomic blocks).

```
Galaxy& asGalaxy();
const Galaxy& asGalaxy() const;
```

These return myself as a Galaxy, overriding `Block::asGalaxy`.

```
const char* domain () const;
```

Return my domain.

```
void setDomain(const char* dom);
```

Set the domain of the galaxy (this may become a protected member in the future).

## 2.3.2 Galaxy protected members

```
void addBlock(Block& b)
```

Add *b* to my list of blocks.

```
void connect(GenericPort& source, GenericPort& destination,
             int numberDelays = 0)
```

Connect sub-blocks with a delay (default to zero delay).

```
void alias(PortHole& galPort, PortHole& blockPort);
void alias(MultiPortHole& galPort, MultiPortHole& blockPort);
```

Connect a Galaxy PortHole to a PortHole of a sub-block, or same for a MultiPortHole.

```
Block* blockWithName (const char* name);
```

Support blockWithName message to access internal block list.

```
void initSubblocks();
void initStateSubblocks();
```

Former: Initialize subblocks only. Latter: initialize states in subblocks only.

### 2.3.3 Galaxy iterators

There are three types of iterators associated with a Galaxy: GalTopBlockIter, GalAllBlockIter, and GalStarIter. The first two iterators return pointers to Block; the final one returns a pointer to Star.

As its name suggests, GalTopBlockIter returns only the Blocks on the top level of the galaxy. GalAllBlockIter returns Blocks at all levels of the hierarchy, in depth-first order; if there is a galaxy inside the galaxy, first it is returned, then its contents are returned. Finally, GalStarIter returns only the atomic blocks in the Galaxy, in depth-first order.

There is also a const form of GalTopBlockIter, called CGalTopBlockIter.

Here is a function that prints out the names of all stars at any level of the given galaxy onto a given stream.

```
void printNames(Galaxy& g,ostream& stream) {
    GalStarIter nextStar(g);
    Star* s;
    while ((s = nextStar++) != 0)
        stream << s->fullName() << "\n";
}
```

## 2.4 Class DynamicGalaxy

A DynamicGalaxy is a type of Galaxy for which all blocks, ports, and states are allocated on the heap. When destroyed, it destroys all of its blocks, ports, and states in a clean manner. There's not much more to it than that: it provides a destructor, class identification functions `isA` and `className`, and little else.

## 2.5 Class InterpGalaxy

InterpGalaxy is derived from DynamicGalaxy. It is the key workhorse for interfacing between user interfaces, such as ptcl or pigi, and the Ptolemy kernel, because it has commands for building structures given commands specified in the form of text strings. These commands add stars and galaxies of given types and build connections between them. InterpGalaxy interacts with the KnownBlock class (see Section 9.1 [class KnownBlock], page 94) to create stars and galaxies, and the Domain class (see Section 9.3 [class Domain], page 97) to create wormholes.

InterpGalaxy differs from other classes derived from Block in that the "class name" (the value returned by `className()`) is a variable; the class is used to create many different "derived classes" corresponding to different topologies.

In order to use InterpGalaxy to make a user-defined galaxy type, a series of commands are executed that add stars, connections, and other features to the galaxy. When a complete galaxy has been designed, the `addToKnownList` member function adds the complete object to the known list, an action that has the effect of adding a new "class" to the system.

InterpGalaxy methods that return an int return 1 for success and 0 for failure. On failure, an appropriate error message is generated by means of the Error class.

### 2.5.1 building structures with InterpGalaxy

The no-argument constructor creates an empty galaxy. There is a constructor that takes a single `const char *` argument specifying the class name (the value to be returned by `className()`. The copy constructor creates another InterpGalaxy with the identical internal structure. There is also an assignment operator that does much the same.

`void setDescriptor(const char*` *dtext*`)`

Set the descriptor. Note that this is public, though the NamedObj function is protected. *dtext* must live as long as the InterpGalaxy does.

`int addStar(const char*` *starname*`, const char*` *starclass*`);`

Add a new star or galaxy with classname *starclass* to this InterpGalaxy, naming the new instance *starname*. The known block list for the current domain is searched to find *starclass*. Returns 1 on success, 0 on failure. On failures, an error message of the form

```
No star/galaxy named 'starclass' in domain 'current-domain'
```

will be produced. The name is a misnomer since *starclass* may name a galaxy or a wormhole.

```
int connect(const char* srcblock, const char* srcport,
            const char* dstblock, const char* dstport,
            const char* delay = 0);
```

This method creates a point-to-point connection between the port *srcport* in the subblock *srcblock* and the port *dstport* in the subblock *dstblock*, with a delay value represented by the expression *delay*. If the delay parameter is omitted there is no delay.

The delay expression has the same form as an initial value for an integer state (class IntState), and is parsed in the same way as an IntState belonging to a subblock of the galaxy would be.

1 is returned for success, 0 for failure. A variety of error messages relating to nonexistent blocks or ports may be produced.

```
int busConnect(const char* srcblock, const char* srcport,
               const char* dstblock, const char* dstport,
               const char* width, const char* delay = 0);
```

This method creates a point-to-point bus connection between the multiport *srcport* in the subblock *srcblock* and the multiport *dstport* in the subblock *dstblock*, with a width value represented by the expression *width* and delay value represented by the expression *delay*. If the delay parameter is omitted there is no delay.

A bus connection is a series of parallel connections: each multiport contains *width* portholes and all are connected in parallel.

The delay and width expressions have the same form as an initial value for an integer state (class IntState), and are parsed in the same way as an IntState belonging to a subblock of the galaxy would be.

1 is returned for success, 0 for failure. A variety of error messages relating to nonexistent blocks or multiports may be produced.

```
int alias(const char* galport, const char* block, const char *blockport);
```

Create a new port for the galaxy and make it an alias for the porthole *blockport* contained in the subblock *block*. Note that this is unlike the Galaxy `alias` method in that this method creates the galaxy port.

```
int addNode(const char* nodename);
```

Create a node for use in netlist-style connections and name it *nodename*.

```
int nodeConnect(const char* blockname, const char* portname,
                const char* node, const char* delay = 0);
```

Connect the porthole named *portname* in the subblock named *blockname* to the node named *node*. Return 1 for success, 0 and an error message for failure.

```
int addState(const char* statename, const char* stateclass,
             const char* statevalue);
```

Add a new state named *statename*, of type *stateclass*, to the galaxy. Its default initial value is given by *statevalue*.

```
int setState(const char* blockname, const char* statename,
             const char* statevalue);
```

Change the initial value of the state named *statename* that belongs to the subblock *blockname* to the string given by *statevalue*. As a special case, if *blockname* is the string `this`, the state belonging to the galaxy, rather than one belonging to a subblock, is changed.

```
int setDomain(const char* newDomain);
```

Change the inner domain of the galaxy to *newDomain*. This is the technique used to create wormholes (that are one domain on the outside and a different domain on the inside). It is not legal to call this function if the galaxy already contains stars.

```
int numPorts(const char* blockname, const char* portname, int numP);
```

Here *portname* names a multiporthole and *blockname* names the block containing it. *numP* portholes are created within the multiporthole; these become ports of the block as a whole. The names of the portholes are formed by appending `#1`, `#2`, etc. to the name of the multiporthole.

### 2.5.2 deleting InterpGalaxy structures

```
int delStar(const char* starname);
```

Delete the instance named *starname* from the current galaxy. Ports of other stars that were connected to ports of *starname* will become disconnected. Returns 1 on success, 0 on failure. On failure an error message of the form

```
No instance of ''starname'' in ''galaxyname''
```

will be produced. The name is a misnomer since *starclass* may name a galaxy or a wormhole.

```
int disconnect(const char* block, const char* port);
```

Disconnect the porthole *port*, in subblock *block*, from whatever it is connected to. This works for point-to-point or netlist connections.

```
int delNode(const char* nodename);
```

Delete the node *nodename*.

### 2.5.3 InterpGalaxy and cloning

```
Block *makeNew() const;
Block *clone() const;
```

For InterpGalaxy the above two functions have the same implementation. An identical copy of the current object is created on the heap.

```
void addToKnownList(const char* outerDomain,Target* innerTarget = 0);
```

This function adds the galaxy to the known list, completing the definition of a galaxy class. The "class name" is determined by the name of the InterpGalaxy (as set by `Block::setBlock` or in some other way). This class name will be returned by the `className` function, both for this InterpGalaxy and for any others produced from it by cloning.

If *outerDomain* is different from the system's current domain (read from class KnownBlock), a wormhole will be created. A wormhole will also be created if *innerTarget* is specified, or if galaxies for the domain *outerDomain* are always wormholes (this is determined by asking the Domain class).

Once `addToKnownList` is called on an InterpGalaxy object, that object should not be modified further or deleted. The KnownBlock class (see Section 9.1 [class KnownBlock], page 94) will manage it from this point on. It will be deleted if a second definition with the same name is added to the known list, or when the program exits.

### 2.5.4 other InterpGalaxy functions

```
const char* className() const
```

Return the current class name (which can be changed). Unlike most other classes, where this function returns the C++ class name, we consider the class name of galaxies built by InterpGalaxy to be variable; it is set by `addToKnownList` and copied from one galaxy to another by the copy constructor or by cloning.

```
void initialize();
```

Overrides `Block::initialize()`. The main extra work is to do variable-parameter initializations, such as delays and bus connections for which the delay value or bus width is an expression with variables.

```
Block* blockWithDottedName(const char* name);
```

Returns a pointer to an inner block, at any depth, whose name matches the specification *name*. For example, `blockWithDottedName("a.b.c")` would look first for a subgalaxy named `"a"`, then within that for a subgalaxy named `"b"`, and finally with that for a subgalaxy named `"c"`, returning either a pointer to the final Block or a null pointer if a match is not found.

## 2.6 Class Runnable

The Runnable class is a sort of mixin class intended to be used with multiple inheritance to create runnable universes and wormholes. It is defined in the file `Universe.h`.

Constructors:

```
Runnable(Target* tar, const char* ty, Galaxy* g);
Runnable(const char* targetname, const char* dom, Galaxy* g);
```

```
void initTarget();
```

This function initializes target and/or generates the schedule.

```
int run();
```

This function causes the object to run, until the stopping condition is reached.

```
virtual void setStopTime(double stamp);
```

This function sets stop time. The default implementation just calls the identical function in the target.

```
StringList displaySchedule();
```

Display schedule, if appropriate (some types of schedulers will return a string saying that compile-time scheduling is not performed, e.g. DE and DDF schedulers).

```
virtual ~Runnable();
```

The destructor deletes the Target.

A Runnable object has the following protected data members:

```
const char* type;
Target* target;
Galaxy* galP;
```

As a rule, when used as one of the baseclasses for multiple inheritance, the `galP` pointer will point to the galaxy provided by the other half of the object.

## 2.7 Class Universe

Class Universe is inherited from both Galaxy and Runnable. It is intended for use in standalone

Ptolemy applications. For applications that use a user interface to dynamically build universes, class InterpUniverse is used instead.

In addition to the Runnable and Galaxy functions, it has:

```
Universe(Target* s,const char* typeDesc);
```

The constructor specifies the target and the universe type.

```
Scheduler* scheduler() const;
```

Returns the scheduler belonging to the universe's target.

```
int run();
```

Return Runnable::Run.

## 2.8 Class InterpUniverse

Class InterpUniverse is inherited from both InterpGalaxy and Runnable. Ptolemy user interfaces build and execute InterpUniverses.

In addition to the standard InterpGalaxy functions, it provides:

```
InterpUniverse (const char* name = "mainGalaxy");
```

This creates an empty universe with no target and the given name. If no name is specified, `mainGalaxy` is the default.

```
int newTarget(const char* newTargName = 0);
```

This creates a target of the given name (from the KnownTarget list), deleting any existing target.

```
const char* targetName() const;
```

Return the name of the current target.

```
Scheduler* scheduler() const;
```

Return the scheduler belonging to the current target (0 if none).

```
Target* myTarget() const;
```

Return a pointer to the current target.

```
int run();
```

Invokes Runnable::run.

```
void wrapup();
```

Invokes wrapup on the target.

# 3 Control of Execution and Error Reporting

The principal classes responsible for control of the execution of the universe are the Target and the Scheduler. The Target has high-level control over what happens when a user types "run" from the interface; Targets take on particular importance in code generation domains where they describe all the features of the target of execution, but they are used to control execution in simulation domains as well.

Targets use Schedulers to control the order of execution of Blocks under their control. In some domains, the Scheduler does almost everything; the Target simply starts it up. In others, the Scheduler determines an execution order and the Target takes care of a many other details, such as generating code in accordance with the schedule, downloading the code to an imbedded processor, and executing it.

The Error class provides a means to format error messages and optionally to halt execution. The interface is always the same, but different user interfaces typically provide different implementations of the methods of this class.

The SimControl class provides a means to register actions for execution during a simulation, as well as facilities to cleanly halt execution on an error.

## 3.1 Class Target

Class Target is derived from class Block (see Section 2.1 [class Block], page 17); at such, it can have states and a parent (the fact that it can also have portholes is not currently used). A Target is capable of supervising the execution of only certain types of Stars; the *starClass* argument in its constructor specifies what type. A Universe or InterpUniverse is run by executing its Target. Targets have Schedulers, which as a rule control order of execution, but it is the Target that is "in control".

A Target can have children that are other Targets; this is used, for example, to represent multiprocessor systems for which code is being generated (the parent target represents the system as a whole, and child targets represent each processor).

### 3.1.1 Target public members

`Target(const char* name, const char* starClass,const char* desc = "");`

This is the signature of the Target constructor. *name* specifies the name of the Target and *desc* specifies its descriptor (these fields end up filling in the corresponding `NamedObj` fields).

The *starClass* argument specifies the class of stars that can be executed by the Target. For example, specifying `DataFlowStar` for this argument means that the Target can run any type of star of this class or a class derived from it. The `isA` function is used to perform the check.

See the description of `auxStarClass` below.

`const char* starType() const;`

Return the supported star class (the *starClass* argument from the constructor).

`Scheduler* scheduler() const;`

Return a pointer to my scheduler.

```
Target* cloneTarget() const;
```

This simply returns the result of the `clone` function as a Target. It is used by the KnownTarget class, for example to create a Target object corresponding to a name specified from a user interface.

```
virtual StringList displaySchedule();
```

The default implementation simply passes this call along to the scheduler; derived classes may modify this.

```
Target* child(int n);
```

Return the *n*th child Target, null if no children or if *n* exceeds the number of children.

```
Target* proc(int n);
```

This is the same as `child` if there are children. If there are no children, an argument of 0 will return a pointer to the object on which it is called, otherwise a null pointer is returned.

```
int nProcs() const;
```

Return the number of processors (1 if no children, otherwise the number of children).

```
virtual int hasResourcesFor(Star& s,const char* extra=0);
```

Determine whether this target has the necessary resources to run the given star. It is virtual in case later necessary. The default implementation uses "resources" states of the target and the star.

```
virtual int childHasResources(Star& s,int childNum);
```

Determine whether a particular child target has resources to run the given star. It is virtual in case later necessary.

```
virtual void setGalaxy(Galaxy& g);
```

Associate a Galaxy with the Target. The default implementation just sets its galaxy pointer `gal` to point to *g*.

`virtual void setStopTime(double` *when*`);`

Set the stopping condition. The default implementation just passes this on to the scheduler.

`virtual void resetStopTime(double` *when*`);`

Reset the stopping condition for the wormhole containing this Target. The default implementation just passes this on to the scheduler. In addition to the action performed by `setStopTime`, this function also does any synchronization required by wormholes.

`virtual void setCurrentTime(double` *now*`);`

Set the current time to *now*.

`virtual int run();`

`virtual void wrapup();`

The following methods are provided for code generation; schedulers may call these. They may move to class CGTarget in a future Ptolemy release.

`virtual void beginIteration(int` *repetitions*`, int` *depth*`);`

Function called to begin an iteration (default version does nothing).

`virtual void endIteration(int` *repetitions*`, int` *depth*`);`

Function called to end an iteration (default version does nothing).

`virtual void writeFiring(Star&` *s*`, int` *depth*`);`

Function called to generate code for the star, with any modifications required by this particular Target (the default version does nothing).

`virtual int commTime(int` *sender*`,int` *receiver*`,int` *nUnits*`, int` *type*`);`

Return the number of time units required to send *nUnits* units of data whose type is the code indicated by *type* from the child Target numbered *sender* to the child target numbered *receiver*. The default implementation returns 0 regardless of the parameters. No meaning is specified at this level for the type codes, as different languages have different types; all that is required is that different types supported by a particular target map into distinct type codes.

```
Galaxy* galaxy();
```

Return my galaxy pointer (0 if it has not been set).

### 3.1.2 Target protected members

```
virtual void setup();
```

This is the main initialization function for the target. It is called by the `initialize` function, which by default initializes the Target states.

The default implementation does the following. The halt flag of SimControl is cleared. Each star is checked to see if its type is supported by the target (because the `isA` function reports that it is one of the supported star classes). If a star does not match this condition an error is reported. If all goes well, then the `setup` member function of the target's Scheduler object is called.

```
void setSched(Scheduler* sch);
```

The target's scheduler is set to *sch*, which must either point to a scheduler on the heap or be a null pointer. Any preexisting scheduler is deleted. Also, the scheduler's `setTarget` member is called, associating the Target with the Scheduler.

```
void delSched();
```

This function deletes the target's scheduler and sets the scheduler pointer to null.

```
void addChild(Target& child);
```

Add *child* as a child target.

```
void inheritChildren(Target* parent, int start, int stop);
```

This method permits two different Target objects to share child Targets. The child targets numbered *start* through *stop* of the Target pointed to by *parent* become the children of this Target (the one on which this method is called). Its primary use is in multi-processor scheduling or code generation, in which some construct is assigned to a group of processors. It has a big disadvantage; the range of child targets must be continuous.

```
void remChildren();
```

Remove the "children" list. This does not delete the child targets.

```
void deleteChildren();
```

Delete all the "children". This assumes that the child Targets were created with `new`.

```
virtual const char* auxStarClass() const;
```

Auxiliary star class: permits a second type of star in addition to supportedStarClass. The default implementation returns a null pointer, indicating no auxiliary star class.

Sorry, there is no present way to support yet a third type.

```
const char* writeDirectoryName(const char* dirName = 0);
```

This method returns a directory name that is intended for use in writing files, particularly for code generation targets. If the directory does not exist, it attempts create it. Returns the fully expanded pathname (which is saved by the target).

```
const char* workingDirectory() const;
```

Return directory name previously set by `writeDirectoryName`.

```
char* writeFileName(const char* fileName = 0);
```

Method to set a file name for writing. *writeFileName* prepends dirFullName (which was set by `writeDirectoryName`) to *fileName* with "/" between. Always returns a pointer to a string in new memory. It is up to the user to delete the memory when no longer needed. If dirFullName or *fileName* is NULL then it returns a pointer to a new copy of the string "/dev/null".

## 3.2 Class Scheduler

Scheduler objects determine the order of execution of Stars. As a rule, they are created and managed by Targets. Some schedulers, such as those for the SDF domain, completely determine the order of execution of blocks before any blocks are executed; others, such as those for the DE domain, supervise the execution of blocks at run time.

The Scheduler class is an abstract base class; you can't have an object of class Scheduler.

All schedulers have a pointer to the Target that controls them as well as to a Galaxy. Usually the Galaxy will be the same one that the Target points to, but this is not a requirement.

The Scheduler constructor just zeros its target, galaxy pointers. The destructor is virtual and do-nothing.

### 3.2.1 Scheduler public members

```
virtual void setGalaxy(Galaxy& g);
```

This function sets the galaxy pointer to point to *g*.

```
Galaxy* galaxy();
```

This function returns the galaxy pointer.

```
virtual void setup() = 0;
```

This function (in derived classes) sets up the schedule. In compile-time schedulers such as those for SDF, a complete schedule is computed; others may do little more than minimal checks.

```
virtual void setStopTime(double limit) = 0;
```

Set the stop time for the scheduler. Schedulers have an abstract notion of time; this determines how long the scheduler will run for.

```
virtual double getStopTime() = 0;
```

Retrieve the stop time.

```
virtual void resetStopTime(double limit);
```

Reset the stopping condition for the wormhole containing this Scheduler. The default implementation simply calls `setStopTime` with the same argument. For some derived types of schedulers, additional actions will be performed as well by derived Scheduler classes.

```
virtual int run() = 0;
```

Run the scheduler until the stop time is reached, an error condition occurs, or it stops for some other reason.

```
virtual void setCurrentTime(double val);
```

Set the current time for the scheduler.

```
virtual StringList displaySchedule();
```

Return the schedule, if this makes sense.

```
double now() const;
```

Return the current time (the value of the protected member `currentTime`).

```
int stopBeforeDeadlocked() const;
```

Return the value of the `stopBeforeDeadFlag` protected member. It is set in timed domains to indicate that a scheduler inside a wormhole was suspended even though it had more work to do.

```
virtual const char* domain() const;
```

Return the domain for this schedule. This method is no longer used and will be removed from future releases; it dates back to the days in which a given scheduler could only be used in one domain.

```
void setTarget(Target& t);
```

Set the target pointer to point to *t*.

```
Target& target ();
```

Return the target.

```
virtual void compileRun();
```

Call code-generation functions in the Target to generate code for a run. In the base class, this just causes an error.

The following functions now forward requests to SimControl, which is responsible for controlling the simulation.

```
static void requestHalt();
```

Calls `SimControl::declareErrorHalt`.

NOTE: `SimControl::requestHalt` only sets the halt bit, not the error bit.

```
static int haltRequested();
```

Calls `SimControl::haltRequested`. Returns TRUE if the execution should halt.

```
static void clearHalt();
```

Calls `SimControl::clearHalt`. Clears the halt and error bits.

### 3.2.2 Scheduler protected members

The following two data members are protected.

```
// current time of the schedule
double currentTime;

// flag set if stop before deadlocked.
// for untimed domain, it is always FALSE.
int stopBeforeDeadlocked;
```

## 3.3 Class Error

Class Error is used for error reporting. While the interfaces to these functions are always the same, different user interfaces provide different implementations: '`ptcl`' connects to the Tcl error reporting mechanism, '`pigi`' pops up windows containing error messages, and '`interpreter`' simply prints messages on the standard error stream. All member functions of Error are static.

There are four "levels" of messages that may be produced by the error facility: `Error::abortRun` is used to report an error and cause execution of the current universe to halt. `Error::error` reports an error. `Error::warn` reports a warning, and `Error::message` prints an information message that is not considered an error.

Each of these four functions is available with two different signatures. For example:

```
static void abortRun (const char*, const char* = 0, const char* = 0);
static void abortRun (const NamedObj& obj, const char*, const char* = 0,
                      const char* = 0);
```

The first form produces the error message by simply concatenating its arguments (the second and third arguments may be omitted); no space is added.

The second form prepends the full name of the *obj* argument, a colon, and a space to the text provided by the remaining arguments. If the implementation provides a marking facility, the object named by *obj* is marked by the user interface (at present, the interface associated with '`pigi`' will highlight the object if its icon appears on the screen).

The remaining static Error functions `error`, `warn`, and `message` have the same signatures as does `abortRun` (there are the same two forms for each function).

In addition, the Error class provides access to the marking facility, if it exists:

```
static int canMark();
```

This function returns TRUE if the interface can mark NamedObj objects (generally true for graphic interfaces), and FALSE if it cannot (generally true for text interfaces).

```
static void mark (const NamedObj& obj);
```

This function marks the object *obj*, if marking is implemented for this interface. It is a no-op if marking is not implemented.

## 3.4 Class SimControl

The SimControl class controls execution of the simulation. It has some global status flags that indicate whether there has been an error in the execution or if a halt has been requested. It also has mechanisms for registering functions to be called before or after star executions, or in response to a particular star's execution, and responding to interrupts.

This class interacts with the Error class (which sets error and halt bits) and the Star class (to permit execution of registered actions when stars are fired). Schedulers and Targets are expected to monitor the SimControl halt flag to halt execution when errors are signaled and halts are requested. Once exceptions are commonplace in C++ implementations, a cleaner implementation could be produced.

### 3.4.1 Access to SimControl status flags.

SimControl currently has four global status bits: the error bit, the halt bit, the interrupt bit, and the poll bit. These functions set, clear, or report on these bits.

```
static void requestHalt ();
```

This function sets the halt bit. The effect is to cause schedulers and targets to cease execution. It is important to note that this function does not alter flow of control; it only sets a flag.

```
static void declareErrorHalt ();
```

This is the same as `requestHalt` except that it also sets the error bit. It is called, for example, by `Error::abortRun`.

```
static int haltRequested ();
```

This function returns true if the halt bit is set, false otherwise. If the poll or interrupt bits are set, it calls handlers for them (see the subsection describing these).

```
static void clearHalt ();
```

This function clears the halt and error flags.

## 3.4.2 Preactions and Postactions

SimControl can register a function that will be called before or after the execution of a particular star, or before or after the execution of all stars. A function that is called before a star is a *preaction*; on that is called after a star is a *postaction*.

The functions that can be registered have take two arguments: a pointer to a Star (possibly null), and a `const char*` pointer that points to a string (possibly null). The type definition

```
typedef void (*SimActionFunction)(Star*,const char*);
```

gives the name SimActionFunction to functions of this type; several SimControl functions take arguments of this form.

```
static SimAction* registerAction(SimActionFunction action, int pre,
 const char* textArg = 0, Star* which = 0);
```

Register a preaction or postaction. If "pre" is TRUE it is a preaction. If *textArg* is given, it is passed as an argument when the action function is called. If *which* is 0, the function will be called unconditionally by `doPreActions` (if it is a preaction) or `doPostActions` (if it is a postaction; otherwise it will only be called if the star being executed has the same address as `which`.

The return value represents the registered action; class SimAction is treated as an "opaque class" (I'm not telling you what is in it) which can be used for `cancel` calls.

```
static int doPreActions(Star * which);
static int doPostActions(Star * which);
```

Execute all pre-actions, or post-actions, for a the particular Star *which*. The *which* pointer is passed to each action function, along with any text argument declared when the action was registered.

Return TRUE if no halting condition arises, FALSE if we are to halt.

```
static int cancel(SimAction* action);
```

Cancel *action*. Warning: argument is deleted.

Future versions will provide more ways of cancelling actions.

### 3.4.3 SimControl interrupts and polling

Features in this section will be used in a new graphic interface; they are mostly untested at this point.

The SimControl class can handle interrupts and can register a polling function that is called for every star execution. It only provides one handler.

```
static void catchInt(int signo = -1, int always = 0);
```

This signal installs a simple interrupt handler for the signal with Unix signal number *signo*. If *always* is true, the signal is always caught; otherwise the signal is not caught if the current status of the signal is that it is ignored (for example, processes running in the background ignore interrupt signals from the keyboard). This handler simply sets the SimControl interrupt bit; on the next call to `haltRequested`, the user-specified interrupt handler is called.

```
static SimHandlerFunction setInterrupt(SimHandlerFunction f);
```

Set the user-specified interrupt handler to *f*, and return the old handler, if any. This function is called in response to any signals specified in `catchInt`.

```
static SimHandlerFunction setPoll(SimHandlerFunction f);
```

Register a function to be called by `haltRequested` if the poll flag is set. Returns old handler if any.

# 4 Interfacing domains – wormholes and related classes

This section describes the classes that implement the mechanism that allows different domains to be interfaced. It is this ability to integrate different domains that sets Ptolemy apart from other systems.

## 4.1 Class Wormhole

A wormhole for a domain is much like a star belonging to that domain, but it contains pointers to a subsystem that operates in a different domain. The interface to that other domain is through a "universal event horizon". The wormhole design, therefore, does not depend on the domain it contains, but only on the domain in which it is used as a block. It must look like a star in that outer domain.

The base Wormhole class is derived from class Runnable (see Section 2.6 [class Runnable], page 31), just like the class Universe (see Section 2.7 [class Universe], page 32). Every member of the Runnable class has a pointer to a component Galaxy (see Section 2.3 [class Galaxy], page 23) and a Target (see Section 3.1 [class Target], page 35).

Like a Universe, a Wormhole can perform the scheduling actions on the component Galaxy. A Wormhole is different from a Universe in that it is not a stand-alone object. Instead, it is triggered from the outer domain to initiate the scheduling. Also, since Wormhole is an abstract baseclass, you cannot create an object of class Wormhole; only derived Wormholes can be created.

Each domain has a derived Wormhole class. For example, the SDF domain has class SDFWormhole. This domain-specific Wormhole is derived from not only the base Wormhole class but also from the domain-specific star class, SDFStar. This multiple inheritance realizes the inherent nature of the Wormhole. First, the Wormhole behaves exactly like a Star from the outer domain (SDF) since it is derived from SDFStar. Second, internally it can encapsulate an entire foreign domain with a separate Galaxy and a separate Target and Scheduler.

### 4.1.1 Wormhole public members

```
void setup();
```

The default implementation calls `initTarget`.

```
int run();
```

This function executes the inside of the wormhole for the appropriate amount of time.

```
const char* insideDomain() const;
```

This function returns the name of the inside domain.

```
void setStopTime(double stamp);
```

This function sets the stop time for the inner universe.

```
Wormhole(Star& self, Galaxy& g, const char* targetName = 0);
Wormhole(Star& self, Galaxy& g, Target* innerTarget = 0);
```

The above two signatures represent the constructors provided for class Wormhole. We never use plain Wormholes; instead we always have objects derived from Wormhole and some kind of Star. For example:

```
class SDFWormhole : public Wormhole, public SDFStar {
public:
    SDFWormhole(Galaxy& g,Target* t) : Wormhole(*this,g,t) {
        buildEventHorizons();
    }
};
```

The first argument to the constructor should always be a reference to the object itself, and represents "the wormhole as a star". The second argument is the inner galaxy. The third argument describes the target of the Wormhole, and may be provided either as a Target object or by name, in which case it is created by using the KnownTarget class.

```
Scheduler* outerSched();
```

This returns a pointer to the scheduler for the outer domain (the one that lives above the wormhole). The scheduler for the inner domain for derived wormhole classes can be obtained from the `scheduler()` method.

### 4.1.2 Wormhole protected members

```
void buildEventHorizons ();
```

This function creates the EventHorizon objects that connect the inner galaxy ports to the outside. A pair of EventHorizons is created for each galaxy port. It is typically called by the constructor for the XXXWormhole, where XXX is the outer domain name.

```
void freeContents();
```

This function deletes the event horizons and the inside galaxy. It is intended to be called from XXXWormhole destructors. It cannot be part of the Wormhole constructor due to an ordering problem (we want to assure that it is called before the destructor for either of XXXWormhole's two baseclasses is called).

```
virtual double getStopTime() = 0;
```

Get the stopping condition for the inner domain. This is a pure virtual function and must be redefined in the derived class.

```
virtual void sumUp();
```

This function is called by `Wormhole::run` after running the inner domain. The default implementation does nothing. Derived wormholes can redefine it to put in any "summing up" work that is required after running the inner domain.

```
Galaxy& gal;
```

The member `gal` is a reference to the inner galaxy of the Wormhole.

## 4.2 Class EventHorizon

Class EventHorizon is another example of a mixin class; EventHorizon has the same relationship to PortHoles as Wormhole has to Stars. The name is chosen from cosmology, representing the point at which an object disappears from the outside universe and enters the interior of a black hole, which can be thought of as a different universe entirely.

As for wormholes, we never consider objects that are "just an EventHorizon". Instead, all objects that are actually used are multiply inherited from EventHorizon and from some type of PortHole class. For each type of domain we require two types of EventHorizon. The first, derived from ToEventHorizon, converts from a format suitable for a particular domain to the "universal form".

The other, derived from FromEventHorizon, converts from the universal form to the domain-specific form.

### 4.2.1 How EventHorizons are used

Generally, EventHorizons are used in pairs to form a connection across a domain boundary between domain XXX and domain YYY. An object of class XXXToUniversal (derived from XXXPortHole and ToEventHorizon) and an object of class YYYFromUniversal (derived from YYYPortHole and FromEventHorizon) are inserted between the ordinary, domain-specific PortHoles. The `far()` member of the XXXToUniversal points to the XXXPortHole; the `ghostAsPort()` member points to the YYYFromUniversal object. Similarly, for the YYYFromUniveral object, `far()` points to the YYYPortHole and `ghostAsPort()` points to the XXXToUniversal object.

These pairs of EventHorizons are created by the `buildEventHorizons` member function of class Wormhole.

### 4.2.2 EventHorizon public members

`EventHorizon(PortHole*` *self*`);`

The constructor for EventHorizon takes one argument, representing (for derived classes that call this constructor from their own), "myself" as a PortHole (a pointer to the PortHole part of the object).

The destructor is declared virtual and does nothing.

```
PortHole* asPort();
```

This returns "myself as a PortHole".

```
PortHole* ghostAsPort();
```

This returns a pointer to the "matching eventhorizon" as a porthole.

`virtual void ghostConnect(EventHorizon&` *to* `);`

This connects another EventHorizon to myself and makes it my "ghost port".

```
virtual int isItInput() const;
virtual int isItOutput() const;
```

Say if I am an input or an output.

```
virtual void setEventHorizon(inOutType inOut, const char* portName,
     Wormhole* parentWormhole, Star* parentStar,
     DataType type = FLOAT, unsigned numTokens = 1 );
```

Sets parameters for the EventHorizon.

```
double getTimeMark();
void setTimeMark(double d);
```

Get and set the time mark. The time mark is an internal detail used for bookkeeping by schedulers.

```
virtual void initialize();
```

```
Scheduler *innerSched();
Scheduler *outerSched();
```

These methods return a pointer to the scheduler that lives inside the wormhole, or outside the wormhole, respectively.

## 4.2.3 EventHorizon protected members

```
void moveFromGhost(EventHorizon& from, int numParticles);
```

Move *numParticles* from the buffer of *from*, another EventHorizon, to mine (the object on which this function is called). This is used to implement `ToEventHorizon::transferData`.

```
CircularBuffer* buffer();
```

Access the myBuffer of the porthole.

```
EventHorizon* ghostPort;
```

This is the peer event horizon.

```
Wormhole* wormhole;
```

This points to the Wormhole I am a member of.

```
int tokenNew;
```

```
double timeMark;
```

TimeMark of the current data, which is necessary for interface of two domains. This may become a private member in future versions of Ptolemy.

## 4.3 Class ToEventHorizon

A ToEventHorizon is responsible for converting from a domain-specific representation to a universal representation. It is derived from EventHorizon.

```
ToEventHorizon(PortHole* p);
```

The constructor simply calls the baseclass constructor, passing along its argument.

```
void initialize();
```

The initialize function prepares the object for execution.

```
void getData();
```

This protected member transfers data from the outside to the universal eventhorizon (myself).

```
void transferData();
```

This protected member transfers data from myself to my peer FromEventHorizon (the ghost-Port).

## 4.4 Class FromEventHorizon

A FromEventHorizon is responsible for converting from a universal representation to a domain-specific representation. It is derived from EventHorizon.

```
FromEventHorizon(PortHole* p);
```

The constructor simply calls the EventHorizon constructor.

```
void initialize();
```

The initialize function prepares the object for execution.

```
void putData();
```

This protected member transfers data from Universal EventHorizon to outside.

```
void transferData();
```

This protected member transfers data from peer event horizon to me.

```
virtual int ready();
```

This is a protected member. By default, it always returns TRUE (1). Derived classes have it return TRUE if the event horizon is "ready" (there is enough data for execution to proceed), and false otherwise.

## 4.5 Class WormMultiPort

The class WormMultiPort, which is derived from MultiPortHole (see Section 5.3 [class MultiPortHole], page 63), exists to handle the case where a galaxy with a multiporthole is imbedded in a wormhole. Its `newPort` function correctly creates a pair of EventHorizon objects when a new port is created in the multiporthole. Instances of this object are created by Wormhole::buildEventHorizons when the inner galaxy has one or more MultiPortHole objects.

# 5 Classes for connections between blocks

This chapter describes the classes that implement connections between blocks. For simulation domains, these classes are responsible for moving objects called Particles (see Section 6.1 [class Particle], page 74) from one Block to another. For code generation domains, the Particles typically only move during scheduling and these objects merely provide information on the topology.

## 5.1 Class GenericPort

The class GenericPort is a base class that provides common elements between class PortHole (see Section 5.2 [class PortHole], page 57) and class MultiPortHole (see Section 5.3 [class MultiPortHole], page 63). Any GenericPort object can be assumed to be either one or the other; we recommend avoiding deriving any new objects directly from GenericPort.

GenericPort is derived from class NamedObj (see Section 1.8 [class NamedObj], page 11).

GenericPort provides several basic facilities: aliases, which specify that another GenericPort should be used in place of this port, types, which specify the type of data to be moved by the port, and typePort, which specifies that this port has the same type as another port. When a GenericPort is destroyed, any alias or typePort pointers are automatically cleaned up, so that other GenericPorts are never left with dangling pointers.

### 5.1.1 GenericPort query functions

```
virtual int isItInput () const;
virtual int isItOutput () const;
virtual int isItMulti () const;
```

Each of the above functions returns TRUE (1) or FALSE (0).

```
StringList print (int verbose = 0) const;
```

Print human-readable information on the wormhole.

```
DataType type () const;
```

Return my DataType. This may be one of the DataType values associated with Particle classes, or the special type `ANYTYPE`, which indicates that the type will be resolved by the `setPlasma` function using information from connected ports and `typePort` pointers.

```
GenericPort* alias() const;
```

Return my alias, or a null pointer if I have no alias. Generally, Galaxy portholes have aliases and Star portholes do not, but this is not a strict requirement.

```
GenericPort* aliasFrom() const;
```

Return the porthole that I am the alias for (a null pointer if none). It is guaranteed that if gp is a pointer to GenericPort and if `gp->alias()` is non-null, then the boolean expression

```
gp->alias()->aliasFrom() == gp
```

is always true.

```
bitWord attributes() const;
```

Return my attributes. Attributes are a series of bits.

```
GenericPort& realPort();
const GenericPort& realPort() const;
```

Return the real port after resolving any aliases. If I have no alias, then a reference to myself is returned.

```
GenericPort* typePort() const;
```

Return another generic port that is constrained to have the same type as me (0 if none). If a non-null value is called, successive calls will form a circular linked list that always returns to its starting point; that is, the loop

```
void printLoop(GenericPort& g) {
        if (g->typePort()) {
                GenericPort* gp = g;
                while (gp->typePort() != g) {
                        cout << gp->fullName() << "\n";
```

```
                                gp = gp->typePort();
                        }
                }
        }
```

is guaranteed to terminate and not to dereference a null pointer.

**`inline int hidden(const GenericPort&`** *p***`)`**

IMPORTANT: **`hidden`** is not a member function of GenericPort, but is a "plain function". It returns TRUE if the port in question has the HIDDEN attribute.

### 5.1.2 other GenericPort public members

**`virtual PortHole& newConnection();`**

Return a reference to a porthole to be used for new connections. Class PortHole uses this one unchanged; MultiPortHole has to create a new member PortHole.

**`GenericPort& setPort(const char*`** *portName*, **`Block*`** *blk*, **`DataType`** *typ*`=FLOAT);`

Set the basic PortHole parameters: the name, parent, and datatype.

**`void inheritTypeFrom(GenericPort&`** *p*`);`

Set up a port for determining the type of `ANYTYPE` connections. typePort pointers form a circular loop.

**`virtual Plasma* setPlasma(Plasma *`***useType* `= NULL) = 0;`

This function associates the appropriate pool of particles, called a Plasma, with the PortHole or MultiPortHole. The effect is also to determine how type conversion will be performed, since the type of a porthole is determined by its associated Plasma.

**`virtual void connect(GenericPort&`** *destination*`,int` *numberDelays*`);`

Connect me with the indicated peer.

```
bitWord setAttributes(const Attribute& attr);
```

Set my attributes (some bits are turned on and others are turned off).

```
void setAlias (GenericPort& gp);
```

Set gp to be my alias. The aliasFrom pointer of gp is set to point to me.

### 5.1.3 GenericPort protected members

```
GenericPort* translateAliases();
```

The above is a protected function. If this function is called on a port with no alias, the address of the port itself is returned; otherwise, `translateAliases(*alias())` is returned.

## 5.2 Class PortHole

PortHole is the means that Blocks use to talk to each other. It is derived from GenericPort; as such, it has a type, an optional alias, and is optionally a member of a ring of ports of the same type connected by `typePort` pointers. It guarantees that `alias()` always returns a PortHole.

In addition, a PortHole has a peer (another port that it is connected to, which is returned by `far()`), a Geodesic (a path along which particles travel between the PortHole and its peer), and a Plasma (a pool of particles, all of the same type). In simulation domains, during the execution of the simulation objects known as Particles traverse a circular path: from an output porthole through a Geodesic to an input porthole, and finally to a Plasma, where they are recirculated back to the input porthole.

Like all NamedObj-derived objects, a PortHole has a parent Block. It may also be a member of a MultiPortHole, which is a logical group of PortHoles.

### 5.2.1 PortHole public members

The constructor sets just about everything to null pointers.

The destructor disconnects the PortHole, and if there is a parent Block, removes itself from the parent's porthole list.

```
PortHole& setPort(const char* portName, Block* parent,
                  DataType type = FLOAT);
```

This function sets the name of the porthole, its parent, and its type.

```
void initialize();
```

This function is responsible for initializing the internal buffers of the porthole in preparation for a run.

```
virtual void disconnect(int delGeo = 1);
```

Remove a connection, and optionally attempt to delete the geodesic. The is set to zero when the geodesic must be preserved for some reason (for example, from the Geodesic's destructor). The Geodesic is deleted only if it is "temporary"; we do not delete "persistent" geodesics when we disconnect them.

```
PortHole* far() const;
```

Return the PortHole we are connected to.

```
void setAlias (PortHole& blockPort);
```

Set my alias to *blockPort*.

```
int atBoundary() const;
```

Return TRUE if this PortHole is at the wormhole boundary (if its peer is an inter-domain connection); FALSE otherwise.

```
virtual EventHorizon* asEH();
```

Return myself as an EventHorizon, if I am one. The baseclass returns a null pointer. EventHorizon objects (objects multiply inherited from EventHorizon and some type of PortHole) will redefine this appropriately.

```
virtual void receiveData();
```

Used to receive data in derived classes. The default implementation does nothing.

```
virtual void sendData();
```

Used to send data in derived classes. The default implementation does nothing.

`Particle& operator % (int` *delay*`);`

This operator returns a reference to a Particle in the PortHole's buffer. A *delay* value of 0 returns the "newest" particle. In dataflow domains, the argument represents the delay associated with that particular particle.

`void setMaxDelay(int` *delay*`);`

Set the maximum delay that past Particles can be accessed – defaults to zero if never called.

```
DataType resolvedType () const;
```

Return the datatype computed by `PortHole::initialize` to resolve type conversions. For example, if an INT output porthole is connected to a FLOAT input porthole, the resolved type (the type of the Particles that travel between the ports) will be FLOAT. A null pointer will be returned if the type has not been set, e.g. before initialization.

```
int numXfer() const;
```

Returns the nominal number of tokens transferred per execution of the PortHole. It returns the value of the protected member `numberTokens`.

```
int numTokens() const;
```

Returns the number of particles on my Geodesic.

```
int numInitDelays() const;
```

Returns the number of initial delays on my Geodesic (the initial tokens, strictly speaking, are only delays in dataflow domains).

```
Geodesic* geo();
```

Return a pointer to my Geodesic.

`Plasma* setPlasma(Plasma *`*useType*` = NULL);`

Initialize the Plasma, which has the effect of resolving the type, since the Particles provided by the Plasma determine the type of data that can be transferred. This function should be protected.

```
int index() const;
```

Return the index value. This is a mechanism for assigning all the portholes in a universe a unique integer index, for use in table-driven schedulers.

```
MultiPortHole* getMyMultiPortHole() const;
```

Return the MultiPortHole that spawned this PortHole, or NULL if there is no such MultiPortHole.

`virtual void setDelay (int `*newDelayValue*`);`

Set the delay value for the connection.

`void adjustDelays(int `*numNewDelays*`);`

Adjust the number of delays on the Geodesic: change the number to *numNewDelays* by using `setDelay` and re-initialize to put the change into effect (name could be better).

```
virtual Geodesic* allocateGeodesic();
```

Allocate a return a Geodesic compatible with this type of PortHole. This may become a protected member in future Ptolemy releases.

### 5.2.2 PortHole protected members

```
Geodesic* myGeodesic;
```

My geodesic, which connects to my peer. Initialized to NULL.

```
PortHole* farSidePort;
```

The port on the far side of the connection. NULL for disconnected ports.

```
Plasma* myPlasma;
```

Pointer to the Plasma where we get our Particles or replace unused Particles. Initialized to NULL.

```
CircularBuffer* myBuffer;
```

Buffer where the Particles are stored. This is actually a buffer of pointers to Particles, not to Particles themselves.

```
int bufferSize;
```

This gives the size of the CircularBuffer to allocate.

```
void getParticle();
```

Get `numberTokens` particles from the Geodesic and move them into my CircularBuffer. Actually, only Particles move. The same number of existing Particles are returned to their Plasma, so that the total number of Particles contained in the buffer remains constant.

```
void putParticle();
```

Move `numberTokens` particles from my CircularBuffer to the Geodesic. Replace them with the same number of Particles from the Plasma.

```
void clearParticle();
```

Clear `numberTokens` particles in the CircularBuffer. Leave the buffer position pointing to the last one.

### 5.2.3 CircularBuffer – a class used to implement PortHole

This class is misnamed; it is not a general circular buffer but rather an array of pointers to Particle that is accessed in a circular manner. It has a pointer representing the current position. This pointer can be advanced or backed up; it wraps around the end when this is done. The class also has a facility for keeping track of error conditions.

The constructor takes an integer argument, the size of the buffer. It creates an array of pointers of that size and sets them all to null. The destructor returns any Particles in the buffer to their Plasma and then deletes the buffer.

```
void reset();
```

Set the access pointer to the beginning of the buffer.

```
Particle** here() const;
```

Return the access pointer. Note the double indirection; since the buffer contains pointers to Particles, the buffer pointer points to a pointer.

```
Particle** next();
```

Advance the pointer one position (circularly) and return the new value.

```
Particle** last();
```

Back up the pointer one position (circularly) and return the new value.

```
void advance(int n);
```

Advance the buffer pointer by *n* positions. This will not work correctly if *n* is larger than the buffer size. *n* is assumed positive.

```
void backup(int n);
```

Back up the buffer pointer by *n* positions. This will not work correctly if *n* is larger than the buffer size. *n* is assumed positive.

```
Particle** previous(int offset) const;
```

Find the position in the buffer *offset* positions in the past relative to the current position. The current position is unchanged. *offset* must not be negative, and must be less than the buffer size, or a null pointer is returned an an appropriate error message is set; this message can be accessed by the `errMsg` function.

```
int size() const;
```

Return the size of the buffer.

```
static const char* errMsg();
```

Return the last error message (currently, only `previous()` sets error messages).

## 5.3 Class MultiPortHole

A MultiPortHole is an organized connection of related PortHoles. Any number of PortHoles can be created within the PortHole; their names have the form *mphname*#1, *mphname*#2, etc., where *mphname* is replaced by the name of the MultiPortHole. When a PortHole is added to the MultiPortHole, it is also added to the porthole list of the Block that contains the MultiPortHole. As a result, a Block that contains a MultiPortHole has, in effect, a configurable number of portholes.

A pair of MultiPortHoles can be connected by a "bus connection". This technique creates *n* PortHoles in each MultiPortHole and connects them all "in parallel".

The MultiPortHole constructor sets the "peer MPH" to 0. The destructor deletes any constituent PortHoles.

### 5.3.1 MultiPortHole public members

```
void initialize();
```

Does nothing.

```
void busConnect (MultiPortHole& peer, int width, int delay = 0);
```

Makes a bus connection with another multiporthole, *peer*, with width *width* and delay *delay*. If there is an existing bus connection, it is changed as necessary; an existing bus connection may be widened, or, if connected to a different peer, all constituent portholes are deleted and a bus is made from scratch.

```
int isItMulti() const;
```

Returns TRUE.

```
MultiPortHole& setPort(const char* portName,
                       Block* parent,DataType type = FLOAT);
```

```
int numberPorts() const;
```

Return the number of PortHoles in the MultiPortHole.

```
virtual PortHole& newPort();
```

Add a new physical port to the MultiPortHole list.

```
MultiPortHole& realPort();
```

Return the real MultiPortHole associated with me, translating any aliases.

```
void setAlias (MultiPortHole &blockPort);
```

Set my alias to *blockPort*.

```
virtual PortHole& newConnection();
```

Return a new port for connections. If there is an unconnected porthole, return the first one; otherwise make a new one.

```
Plasma* setPlasma(Plasma *useType = NULL);
```

Sets the Plasma for all the portholes in the multiporthole.

### 5.3.2 MultiPortHole protected members

`PortList ports;`

The list of portholes (should be protected).

`const char* newName();`

This function generates names to be used for contained PortHoles. They are saved in the hash table provided by the `hashstring` function (see Section 1.3 [utility functions], page 2).

`PortHole& installPort(PortHole& p);`

This function adds a newly created port to the multiporthole. Derived MultiPortHole classes typically redefine `newPort` to create a porthole of the appropriate type, and then use this function to register it and install it.

`void delPorts();`

This function deletes all contained portholes.

## 5.4 AutoFork and AutoForkNode

AutoForks are a method for implementing netlist-style connections. An AutoForkNode is a type of Geodesic built on top of AutoFork. The classes are separate to allow a "mixin approach", so that if a domain requires special actions in its Geodesics, these special actions can be written only once and be implemented in both temporary and permanent connections.

The implementation technique used is to automatically insert a Fork star to allow the n-way connection; this Fork star is created by invoking `KnownBlock::makeNew("Fork")`, which works only for domains that have a fork star.

### 5.4.1 Class AutoFork

An AutoFork object has an associated Geodesic and possibly an associated Fork star (which it creates and deletes as needed). It is normally used in a multiply inherited object, inherited from AutoFork and some kind of Geodesic; hence the associated Geodesic is the object itself.

The constructor for class AutoFork takes a single argument, a reference to the Geodesic. It sets the pointer to the fork star to be null.

The destructor removes the fork star, if one was created.

There are two public member functions, `setSource` and `setDest`.

```
PortHole* setSource(GenericPort& port, int delay = 0);
```

If there is already an originating port for the geodesic, this method returns an error. Otherwise it connects it to the node.

```
PortHole* setDest(GenericPort& port, int alwaysFork = 0);
```

This function may be used to add any number of destinations to the port. Normally, when there is more than one output, a Fork star is created and inserted to support the multi-way connection, but if there is only one output, a direct connection is used. However, if *alwaysFork* is true, a Fork is inserted even for the first output. When the fork star is created, it is inserted in the block list for the parent galaxy (the parent of the geodesic).

### 5.4.2 Class AutoForkNode

Class AutoForkNode is multiply inherited from Geodesic and AutoFork.

This class redefines `isItPersistent` to return TRUE, and redefines the `setSourcePort` and `setDestPort` functions to call the `setSource` and `setDest` functions of AutoFork. The exact same form could be used to generate other types of auto-forking nodes (that is, this class could have been done with a template).

## 5.5 Class ParticleStack

ParticleStack is an efficient baseclass for the implementation of structures that organize Particles. As Particles have a link field, ParticleStack is simply implemented as a linked list of Particles.

Strictly speaking, a dequeue is implemented; particles can be inserted from either end. ParticleStack has some odd attributes; it is designed for very efficient implementation of Geodesic and Plasma to move around large numbers of Particle objects very efficiently.

```
ParticleStack(Particle* h);
```

The constructor takes a Particle pointer. If it is a null pointer an empty ParticleStack is created. Otherwise the stack has one particle. Adding a Particle to a ParticleStack modifies that Particle's link field; therefore a Particle can belong to only one ParticleStack at a time.

```
~ParticleStack();
```

The destructor deletes all Particles EXCEPT for the last one; we do not delete the last one because it is the "reference" particle (for Plasma) and is normally not dynamically created (this code may be moved in a future release to the Plasma destructor, as this behavior is needed for Plasma and not for other types of ParticleStack).

```
void put(Particle* p);
```

Push *p* onto the top (or head) of the ParticleStack.

```
Particle* get();
```

Pop the particle off the top (or head) of the ParticleStack.

```
void putTail(Particle* p);
```

Add *p* at the bottom (or tail) of the ParticleStack.

```
int empty() const;
```

Return TRUE (1) if the ParticleStack is empty, otherwise 0.

```
int moreThanOne() const;
```

Return TRUE (1) if the ParticleStack has two or more particles, otherwise 0. This is provided to speed up the derived class Plasma a bit.

```
void freeup();
```

Returns all Particles on the stack to their Plasma (the allocation pool for that particle type).

There is one protected data member:

```
Particle* head;
```

This is the head of the list (or the top of the stack).

## 5.6 Class Geodesic

A Geodesic implements the connection between a pair, or a larger collection, of PortHoles. A Geodesic may be temporary, in which case it is deleted when the connection it implements is broken, or it can be permanent, in which case it can live in disconnected form. As a rule, temporary geodesics are used for point-to-point connections and permanent geodesics are used for netlist connections. In the latter case, the Geodesic has a name and is a member of a galaxy; hence, Geodesic is derived from NamedObj (see Section 1.8 [class NamedObj], page 11).

The baseclass Geodesic, which is temporary, suffices for most simulation and code generation domains. In fact, in a number of these domains it contains unused features, so it is perhaps too "heaviweight" an object. A Geodesic contains a ParticleStack member which is used as a queue for movement of Particles between two portholes; it also has an originating port and a destination port.

A Geodesic can be asked to have a specific number of initial particles. When initialized, it creates that number of particles in its ParticleStack; these particles are obtained from the Plasma of the originating port (so they will be of the correct type).

### 5.6.1 Geodesic public members

```
virtual PortHole* setSourcePort (GenericPort &src, int delay = 0);
```

Set the source port and the number of initial particles. The actual source port is determined by calling `newConnection` on *src*; thus if *src* is a MultiPortHole, the connection will be made to some port withing that MultiPortHole, and aliases will be resolved. The return value is the "real porthole" used.

In the default implementation, if there is already a destination port, any preexisting connection is broken and a new connection is completed.

```
virtual PortHole* setDestPort (GenericPort &dest);
```

Set the destination port to *dest*`.newConnection()`. The return value is the "real porthole" used.

In the default implementation, if there is already a source port, any preexisting connection is broken and a new connection is completed.

```
virtual int disconnect (PortHole & p);
```

In the default implementation, if *p* is either the source port or the destination port, both the source port and destination port are set to null. This is not enough to break a connection; as a rule, `disconnect` should be called on the porthole, and that method will call this one as part of its work.

```
virtual void setDelay (int newDelay);
```

Modify the delay (number of initial tokens) of a connection. The default implementation simply changes a count.

```
virtual int isItPersistent() const;
```

Return true if the Geodesic is persistent (may exist in a disconnected state) and false otherwise. The default implementation returns false.

```
PortHole* sourcePort () const;
PortHole* destPort () const;
```

Return my source and destination ports, respectively.

```
virtual void initialize();
```

In the default implementation, this function initializes the number of Particles to that given by the numInitialParticles field (the value returned by `numInit()`; these Particles are obtained from the Plasma (allocation pool) for the source port. The particles will have zero value for numeric particles, and will hold the "empty message" for message Particles.

```
void put(Particle* p);
```

Put a particle into the Geodesic (using a FIFO discipline).

```
Particle* get();
```

Retrieve a particle from the Geodesic (using a FIFO discipline). Return a null pointer if the Geodesic is empty.

```
void pushBack(Particle* p);
```

Push a Particle back into the Geodesic (onto the front of the queue, instead of onto the back of the queue as `put` does).

```
int size() const;
```

Return the number of Particles on the Geodesic at the current time.

```
int numInit() const;
```

Return the number of initial particles. This call is valid at any time. Immediately after `initialize`, `size` and `numInit` return the same value (and this should be true for any derived Geodesic as well), but this will not be true during execution (where `numInit` stays the same and `size` changes).

```
StringList print(int verbose) const;
```

Print information on the Geodesic, overrides `NamedObj` function.

```
virtual void incCount(int);
virtual void decCount(int);
```

These methods are available for schedulers such as the SDF scheduler to simulate a run and keep track of the number of particles on the geodesic. `incCount` increases the count, `decCount` decreases it, They are virtual to allow additional bookkeeping in derived classes.

### 5.6.2 Geodesic protected members

```
void portHoleConnect();
```

This function completes a connection once the originating and destination ports are set up.

```
PortHole *originatingPort;
PortHole *destinationPort;
```

These protected members point to my neighbors.

## 5.7 Class Plasma

Class Plasma is a pool for particles. It is derived from ParticleStack (see Section 5.5 [class ParticleStack], page 67). Rather than allocating Particles as needed with `new` and freeing them with `delete`, we instead provide an allocation pool for each type of particle, so that very little dynamic memory allocation activity will take place during simulation runs.

All Plasma objects known to the system are linked together. As a rule, there is one Plasma for each type of particle; however, each of these objects is of type Plasma, not a derived type. At all times, a Plasma has at least one Particle in it; that Particle's virtual functions are used to clone other particles as needed, determine the type, etc.

The constructor takes one argument, a reference to a Particle. It creates a one-element ParticleStack, and links the Plasma into a linked list of all Plasma objects.

The put function (for putting a particle into the Plasma) adds a particle to the Plasma's ParticleStack. As a rule, it should not be used directly; the Particle's `die` method will automatically add it to the right Plasma (future releases may protect this method to prevent its general use).

```
Particle* get();
```

This function gets a Particle from the Plasma, creating a new one if the Plasma has only one Particle on it (we never give away the last Particle).

```
DataType type();
```

Returns the type of the particles on the list (obtained by asking the head Particle).

```
static Plasma* getPlasma(DataType type);
```

Searches the list of Plasmas for one whose type matches the argument, and returns a pointer to it. A null pointer is returned if there is no match.

## 5.8 Class ParticleQueue

Class ParticleQueue implements a queue of Particles. It uses a member of class ParticleStack to store the particles; it is not implemented by deriving from ParticleStack. It can implement a queue with finite or unlimited capacity.

Rather than placing user-supplied Particles on the queue and removing them directly, it takes over the responsibility for memory management by allocating its own Particles from the Plasma and returning them as needed. When a user puts a Particle into the queue, the value of the Particle is copied (with the Particle `clone` method); similarly, when a user gets a Particle from the queue, he or she supplies a Particle to received the copied value. The advantage of this is that the user need not worry about lifetimes of Particles – when to create them, when it is safe to return them to the Plasma or delete them.

The ParticleQueue default constructor forms an empty, unlimited capacity queue. There is also a constructor of the form

```
ParticleQueue(unsigned int cap);
```

This creates a queue that can hold at most *cap* particles.

The destructor returns all Particles in the queue to their Plasma.

```
int empty() const;
```

Return TRUE if the queue is empty, else FALSE.

```
int full() const;
```

Return TRUE if the length equals the capacity, else FALSE.

```
unsigned int capacity() const;
```

Return the queue's capacity. If unlimited, the largest possible unsigned int on the machine will be returned.

```
unsigned int length() const;
```

Return the number of particles in the queue.

```
int putq(Particle& p);
```

Put a copy of particle *p* into the queue, if there is room. Returns TRUE on success, FALSE if the queue is already at capacity.

```
int getq(Particle& p);
```

Get a particle from the queue, and copy it into the user-supplied particle *p*. This returns TRUE on success, FALSE (and *p* is unaltered) if the queue is empty.

```
void setCapacity(int sz);
```

Modify the capacity to *sz*, if *sz* is positive or zero. If negative, the capacity becomes infinite.

```
void initialize();
```

Free up the queue contents. Particles are returned to their pools and the queue becomes empty.

```
void initialize(int n);
```

Equivalent to `initialize()` followed by `setCapacity(n)`.

## 5.9 Classes for Galaxy ports

Class GalPort is derived from class PortHole (see Section 5.2 [class PortHole], page 57).

Class GalMultiPort is derived from class MultiPortHole (see Section 5.3 [class MultiPortHole], page 63).

These classes are used by InterpGalaxy (see Section 2.5 [class InterpGalaxy], page 27), and in other places, to create galaxy ports and multiports that are aliased to some port of a member block. The constructor for each of these classes takes one argument, the interior port that is to be the alias. The `isItInput` and `isItOutput` functions are implemented by forwarding the request to the alias.

# 6 Particles and Messages

## 6.1 Class Particle

A Particle is a little package that contains data; they represent the principal communication technique that blocks use to pass results around. They move through PortHoles and Geodesics; they are allocated in pools called Plasmas. The class Particle is an abstract base class; all real Particle objects are really of some derived type.

All Particles contain a link field that allows queues and stacks of Particles to be manipulated efficiently (class ParticleStack is a base class for everything that does this).

Particles also contain virtual operators for loading and accessing the data in various forms; these functions permit automatic type conversion to be easily performed.

## 6.2 Particle public members

```
virtual DataType type() const = 0;
```

Return the type of the particle. DataType is actually just a typedef for `const char*`, but when we use DataType, we treat it as an abstract type. Furthermore, two DataType values are considered the same if they compare equal, which means that we must assure that the same string is always used to represent a given type.

```
virtual operator int () const = 0;
virtual operator float () const = 0;
virtual operator double () const = 0;
virtual operator Complex () const = 0;
```

These are the virtual casting functions, which convert the data in the Particle into the desired form. The arithmetic Particles support all these functions cleanly. Message particles may return errors for some of these functions (they must return a value, but may also call `Error::abortRun`.

```
virtual StringList print () const = 0;
```

Return a printable representation of the Particle's data.

```
virtual void initialize() = 0;
```

This function zeros the Particle (where this makes sense), or initializes it to some default value.

```
virtual void operator << (int arg) = 0;
virtual void operator << (double arg) = 0;
virtual void operator << (const Complex& arg) = 0;
```

These functions are, in a sense, the inverses of the virtual casting operators. They load the particle with data from *arg*, performing the appropriate type conversion.

```
virtual Particle& operator = (const Particle& arg) = 0;
```

Copy a Particle. As a rule, we permit this only for Particles of the same type, and otherwise assert an error.

```
virtual int operator == (const Particle&) = 0;
```

Compare two particles. As a rule, Particles will be equal only if they have the same type, and, in a sense that is separately determined for each type, the same value.

```
virtual Particle* clone() const = 0;
```

Produce a second, identical particle (as a rule, one is obtained from the Plasma for the particle if possible).

```
virtual Particle* useNew() const = 0;
```

This is similar to clone, except that the particle is allocated from the heap rather than from the Plasma.

```
virtual void die() = 0;
```

Return the Particle to its Plasma.

```
virtual void getMessage (Envelope&);
virtual void accessMessage (Envelope&) const;
virtual void operator << (const Envelope&);
```

These functions are used to implement the Message interface. The default implementation returns errors for them; it is only if the Particle is really a MessageParticle that they successfully send or receive a Message from the Particle.

## 6.3 Arithmetic Particle classes

There are three standard arithmetic Particle classes: IntParticle, FloatParticle, and Complex-Particle. As their names suggest, each class adds to Particle a private data member of type int, double (not float!), and class Complex, respectively.

When a casting operator or "<<" operator is used on a particle of one of these types, a type conversion may take place. If the type of the argument of cast matches the type of the particle's data, the data is simply copied. If the requested operation involves a "widening" conversion (int to float, double, or Complex; float to double or Complex; double to Complex), the "obvious" thing happens. Conversion from double to int rounds to the nearest integer; conversion from Complex to double returns the absolute value (not the real part!), and Complex to int returns the absolute value, rounded to the nearest integer.

`initialize` for each of these classes sets the data value to zero (for the appropriate domain).

The DataTypes returned by these Particle types are the global symbols INT, FLOAT, and COMPLEX, respectively. They have the string values "INT", "FLOAT", and "COMPLEX".

## 6.4 The Heterogeneous Message Interface

The heterogeneous message interface is a mechanism to permit messages of arbitrary type (objects of some derived type of class Message) to be transmitted by blocks. Because these messages may be very large, facilities are provided to permit many references to the same Message; Message objects are "held" in another class called Envelope. As the name suggests, Messages are transferred in Envelopes. When Envelopes are copied, both Envelopes refer to the same Message. A Message will be deleted when the last reference to it disappears; this means that Messages must always be on the heap.

So that Messages may be transmitted by portholes, there is a class MessageParticle whose data field is an Envelope. This permits it to hold a Message just like any other Envelope object.

### 6.4.1 Class Envelope

class Envelope has two constructors. The default constructor constructs an "empty" Envelope (in reality, the envelope is not empty but contains a special "dummy message" – more on this later). There is also a constructor of the form

```
Envelope(Message& data);
```

This constructor creates an Envelope that contains the Message *data*, which MUST have been allocated with `new`.

Message objects have reference counts; at any time, the reference count equals the number of Envelope objects that contain (refer to) the Message object. When the reference count drops to zero (because of execution of a destructor or assignment operator on an Envelope object), the Message will be deleted.

Class Envelope defines an assignment operator, copy constructor, and destructor. The main work of these functions is to manipulate reference counts. When one Envelope is copied to another, both Envelopes refer to the same message.

```
int empty() const;
```

Return TRUE if the Envelope is "empty" (points to the dummy message), FALSE otherwise.

```
const Message* myData() const;
```

Return a pointer to the contained Message. This pointer must not be used to modify the Message object, since other Envelopes may refer to the same message.

```
Message* writableCopy();
```

This method produces a writable copy of the contained Message, and also zeros the Envelope (sets it to the empty message). If this Envelope is the only Envelope that refers to the message, the return value is simply the contained message. If there are multiple references to the message, the `clone` method is called on the Message, making a duplicate, and the duplicate is returned.

The user is now responsible for memory management of the resulting Message. If it is put into another Envelope, that Envelope will take over the responsibility, deleting the message when there is no more need for it. If it is not put into another Envelope, the user must make sure it is deleted somehow, or else there will be a memory leak.

`int typeCheck(const char*` *type*`) const;`

This member function asks the question "is the contained Message of class *type*, or derived from *type*"? It is implemented by calling `isA` on the Message. Either TRUE or FALSE is returned.

`const char* typeError(const char*` *expected*`) const;`

This member function may be used to format error messages for when one type of Message was expected and another was received. The return value points to a static buffer that is wiped out by subsequent calls.

```
const char* dataType() const;
int asInt() const;
double asFloat() const;
Complex asComplex() const;
StringList print() const;
```

All these methods are "passthrough methods"; the return value is the result of calling the identically named function on the contained Message object.

### 6.4.2 Class Message

Message objects can be used to carry data between blocks. Unlike Particles, which must all be of the same type on a given connection, connections that pass Message objects may mix message objects of many types on a given connection. The tradeoff is that blocks that receive Message objects must, as a rule, type-check the received objects.

The baseclass for all messages, named Message, contains no data, only a reference count (accordingly, all derived classes have a reference count and a standard interface). The reference count counts how many Envelope objects refer to the same Message object.

The constructor for Message creates a reference count that lives on the heap. This means that the reference count is non-const even when the Message object itself is const.

The copy constructor for Message ignores its argument and creates a new Message with a new reference count. This is necessary so that no two messages will share the same reference count.

The destructor, which is virtual, deletes the reference count.

The following Message functions must be overriden appropriately in any derived class:

```
virtual const char* dataType() const;
```

This function returns the type of the Message. The default implementation returns "DUMMY".

```
virtual Message* clone() const;
```

This function produces a duplicate of the object it is called on. The duplicate must be "good enough" so that applications work the same way whether the original Message or one produced by `clone()` is received. A typical strategy is to define the copy constructor for each derived Message class and write something like

```
Message* MyMessage::clone() const { return new MyMessage(*this);}
```

```
    virtual int isA(const char*) const;
```

The isA function returns true if given the name of the class or the name of any baseclass. Exception: the baseclass function returns FALSE to everything (as it has no data at all). A macro ISA_FUNC is defined to automate the generation of implementations of derived class isA functions; it is the same one as that used for the NamedObj class.

The following methods may optionally be redefined.

```
    virtual StringList print() const;
```

This method returns a printable representation of the Message. The default implementation returns a message like

```
    Message class <type>: no print method
```

where *type* is the message type as returned by the dataType function.

```
    virtual int asInt() const;
    virtual double asFloat() const;
    virtual Complex asComplex() const;
```

These functions represent conversions of the Message data to an integer, a floating point value, and a complex number, respectively. Usually such conversions do not make sense; accordingly, the default implementations generate an error message (using the protected member function errorConvert) and return a zero of the appropriate type. If a conversion does make sense, they may be overriden by a method that does the appropriate conversion. These methods will be used by the MessageParticle class when an attempt is made to read a MessageParticle in a numeric context.

One protected member function is provided:

```
    int errorConvert(const char* cvttype) const;
```

This function invokes Error::abortRun with a message of the form

```
    Message class <msgtype>: invalid conversion to cvttype
```

where *msgtype* is the type of the Message, and *cvttype* is the argument.

### 6.4.3 Class MessageParticle

MessageParticle is a derived type of Particle (see Section 6.1 [class Particle], page 74) whose data field is an Envelope; accordingly, it can transport Message objects.

MessageParticle defines no new methods of its own; it only provides behaviors for the virtual functions defined in class Particle. The most important such behaviors are as follows:

```
void operator << (const Envelope& env);
```

This method loads the Message contained in *env* into the Envelope contained in the MessageParticle. Since the Envelope assignment operator is used, after execution of this method both *env* and the MessageParticle refer to the message, so its reference count is at least 2.

```
void getMessage(const Envelope& env);
```

This method loads the message contained in the MessageParticle into the Envelope *env*, and removes the message from the MessageParticle (so that it now contains the dummy message). If *env* previously contained the only reference to some other Message, that previously contained Message will be deleted.

```
void accessMessage(const Envelope& env);
```

`accessMessage` is the same as `getMessage` except that the message is not removed from the MessageParticle. It can be used in situations where the same Particle will be read again. We recommend that `getMessage` be used where possible, especially for very large message objects, so that they are deleted as soon as possible.

## 6.5 Example Message types

The kernel provides two simple sample message types for transferring arrays of data. They are almost identical except that one holds an array of integers and the other holds an array of single precision floating point data. The array contents live on the heap. Each is derived from class Message.

Each provides a public data member that points to the data. As a rule, we recommend against public data members for classes, but an exception was made in this case, perhaps unwisely.

This section will describe the interface of the FloatVecData class. The interface for IntVecData is almost identical.

Three constructors are provided:

```
FloatVecData(int len);
```

This form creates an uninitialized array of length *len* in the FloatVecData object. Since the pointer to the data is public the array may easily be filled in.

```
FloatVecData(int len,const float *srcData);
```

This form creates an array of length *len* and initializes it with *len* elements from *srcData*.

```
FloatVecData(int len,const double *srcData);
```

This form is the same, except that the source data is double precision (it is converted to single precision). This is the only function for which an analogous function does not exist in IntVecData (an IntVecData can only be initialized from an integer array).

An appropriate copy constructor, assignment operator, and destructor are defined.

```
int length() const;
```

Return the length of the array.

```
float *data;
```

Public data member; points to the array. It is permissible to read or assign the *len* elements starting at `data`; the effect of altering the `data` pointer itself is undefined.

```
const char* dataType() const;
```

Returns the string `"FloatVecData"`.

```
int isA(const char* type) const;
```

TRUE for *type* equal to "FloatVecData", otherwise false.

```
StringList print() const;
```

Returns a comma-separated list of elements enclosed in curly braces.

```
Message* clone() const;
```

Creates an identical copy with `new`.

# 7 The incremental linker

The incremental linker permits user written code to be added to the system at runtime. Two different mechanisms are provided, called a temporary link and a permanent link.

In either case, code is linked using the incremental linking facilities of the Unix linker, the new code is read into the Ptolemy executable, and symbols corresponding to C++ global constructors are located and called. This means that such code is expected to register objects on Ptolemy's known lists (e.g. KnownBlock, KnownState, or KnownTarget) so that new classes become usable.

*Warning:* if the executable containing the Linker class is stripped, the incremental linker will not work!

### 7.0.1 Temporary vs. Permanent Incremental Linking

Code that is linked in by the "temporary link" technique does not alter the symbol table in use. For that reason, subsequent incremental links, whether temporary or permanent, cannot "see" any code that was linked in by previous temporary links. The advantage is that the same symbols (for example, a Ptolemy star definition) may be redefined, which is useful in code development, as buggy star definitions can be replaced by valid ones without exiting Ptolemy.

Code that is linked in by the "permanent link" method has the same status as code that was linked into the original executable. A permanent link creates or replaces the '`.pt_symtable`' file in the directory in which Ptolemy was started. This file contains the current symbol table for use by subsequent links, temporary or permanent. This file is deleted when the Ptolemy process exits normally. It is left around when the process crashes, as it is useful for debugging (as it contains symbols for object files that were incrementally linked using the permanent method as well as those in the original executable).

### 7.0.2 Linker public members

```
static void init(const char* execName);
```

This function initializes the linker module by telling it where the executable for this program is. For most purposes, passing it the value of `argv[0]` passed to the `main` function will suffice.

```
static int linkObj(const char* objName);
```

Link in a single object module using the temporary link mechanism (this entry point is provided for backward compatibility).

```
static int multiLink(const char* args, int permanent);
static int multiLink(int argc, char** argv);
```

Both of these functions give access to the main function for doing an incremental link. They permit either a temporary or a permanent link of multiple files; flags to the Unix linker such as `-l` to specify a library or `-L` to specify a search directory for libraries are permitted. For the first form, *args* are passed as part of a linker command that is expanded by the Unix shell. A permanent link is performed if *permanent* is true (nonzero); otherwise a temporary link is performed.

The second form is provided for ease of interfacing to the Tcl interpreter, which likes to pass arguments to commands in this style. In this case, *argv*`[0]` indicates the type of link: if it begins with the character `p`, a permanent link is performed; otherwise a temporary link is performed. The remaining arguments are concatentated (separated by spaces) and appear in the argument to the Unix linker.

```
static int isActive();
```

This function returns TRUE if the linker is currently active (so objects can be marked as dynamically linked by the known list classes). Actually the flag it returns is set while constructors or other functions that have just been linked are being run.

```
static int enabled();
```

Returns true if the linker is enabled (it is enabled by calling `Linker::init` if that function returns successfully.

```
static const char* imageFileName();
```

Return the fully-expanded name of the executable image file (set by `Linker::init`.

```
static void setDefaultOpts(const char* newValue);
static const char* defaultOpts();
```

These functions set or return the linker's default options, a set of flags appended to the end of the command line by all links.

### 7.0.3 Linker implementation

This section is intended to assist those that attempt to port the Linker module to other platforms.

The Linker class is implemented in three files: '`Linker.h`', specifying the class interface, '`Linker.cc`', specifying the implementation, and '`Linker.sysdep.h`', specifying all the machine dependent parts of the implementation.

The Linker class currently works on the Sun-3 or Sun-4 running g++, the Sun-4 running Sun's `cfront` port, and Vaxes and DecStations running Ultrix. Some work has been done to port it to the HP-PA architecture; this is not yet complete.

The intent is to structure the code in such a way that no `#ifdefs` appear in '`Linker.cc`'; they should all be in '`Linker.sysdep.h`'.

The linker reads all new code into a pre-existing large array, rather than creating blocks of the right size with `new`, because the right size is not known in advance but a starting location must, as a rule, be passed to the loader in advance. This means that there is a wired-in limit to how much code can be linked in. The symbol `LINK_MEMORY`, which is set to one megabyte by default, is easily changed if required.

Here are the steps taken by the linker to do its work:

Align the memory as required.

Form the command line and execute the Unix linker. Only certain flags in the command line will be system-dependent.

Read in the object file. This is heavily system-dependent.

Make the read-in text executable. On most systems this is a do-nothing step.

Invoke constructors in the newly read in code. Constructors are found by use of the 'nm' program; the output is parsed to search for constructor symbols, whose form depends on the compiler used.

If this is a permanent link, copy the linker output to file '`.pt_symtable`'; otherwise delete it.

# 8 Parameters and States

A State is a data-structure associated with a block, used to remember data values from one invocation to the next. For example, the gain of an automatic gain control is a state. A state need not be dynamic; for instance, the gain of fixed amplifier is a state. A parameter is the initial value of a state.

A State actually has two values: the initial value, which is always a character string, and a current value, whose type is different for each derived class of State: integer for IntState, an array of real values for FloatArrayState, etc.

In addition, states have attributes, which represent logical properties the state either has or does not have.

## 8.1 Class State

Class State is derived from class NamedObj (see Section 1.8 [class NamedObj], page 11).

The State baseclass is an abstract class; you cannot create a plain State. The baseclass contains the initial value, which is always a `const char*`; the derived classes are expected to provide current values of appropriate type.

The constructor for class State sets the initial value to a null pointer, and sets the state's attributes to a value determined by the constant AB_DEFAULT. The destructor does nothing extra.

### 8.1.1 State public members

```
State& setState(const char* stateName, Block* parent,
                const char* initValue, const char* desc = NULL);
```

This function sets the name, parent, initial value, and optionally the descriptor for a state. The character strings representing the initial value and descriptor must outlive the State.

```
State& setState(const char* stateName, Block* parent,
                const char* initValue, const char* desc,
                Attribute attr);
```

This function is the same as the other `setState`, but it also sets attributes for the state. The Attribute object represents a set of attribute bits to turn on or off.

```
void setInitValue(const char* valueString);
```

This function sets the initial value to *valueString*. This string must outlive the State.

```
const char* initValue () const;
```

Return the initial value.

```
virtual const char* type() const = 0;
```

Return the type name (for use in user interfaces, for example). When states are created dynamically (by the KnownState or InterpGalaxy class), it is this name that is used to specify the type.

```
virtual int size() const;
```

Return the size (number of distinct values) in the state. The default implementation returns 1. Array state types will return the number of elements.

```
virtual int isArray() const;
```

Return TRUE if this state is an array, false otherwise. The default implementation returns false.

```
virtual void initialize() = 0;
```

Initialize the state. The `initialize` function for a state is responsible for parsing the initial value string and setting the current value appropriately; errors are signaled using the `Error::abortRun` mechanism.

```
virtual StringList currentValue() const = 0;
```

Return a string representation of the current value.

```
void setCurrentValue(const char* newval);
```

Modify the current value, in a type-independent way. Notice that this function is not virtual. It exploits the semantics of `initialize` to set the current value using other functions; the initial value is not modified (it is saved and restored).

```
virtual State* clone() const = 0;
```

Derived state classes override this method to create an identical object to the one the method is called on.

```
StringList print(int verbose) const;
```

Output all info. This is NOT redefined for each type of state.

```
bitWord attributes() const;
```

Return my attribute bits.

```
bitWord setAttributes(const Attribute& attr);
bitWord clearAttributes(const Attribute& attr);
```

Set or clear attributes.

## 8.1.2 The State parser and protected members

Most of class State's protected interface consists of a simple recursive-descent parser for parsing integer and floating expressions that appear in the initial value string.

class ParseToken represents tokens for this parser; it contains a token type (an integer code) and a token value, which is a union that represents either a character value, a string value, an integer value, a double value, a Complex value, or a State value (for use when the initializer references another state). Token types are equal to the ASCII character value for single-character tokens; other possible token values are:

`T_EOF` for end of file, `T_ERROR` for error, `T_Float` for a floating value, `T_Int` for an integer value, `T_ID` for a reference to a state, and `T_STRING` for a string value. For most of these, the token value holds the appropriate value.

Most derived State classes use this parser to provide uniformity of syntax and error reporting; however, it is not a requirement to use it.

Derived State classes are expected to associate a Tokenizer object with their initial value string; the functions provided here can then be used to parse expressions appearing in that string.

```
ParseToken getParseToken(Tokenizer& tok, int wantedType= T_Float);
```

This function obtains the next token from the input stream associated with the Tokenizer. If there is a pushback token, that token is returned instead. If it receives a '<' token, it assumes that the next whitespace-delimited string is a file and uses Tokenizer's include file capability to cause it to read from that file.

It returns a T_EOF token on end of file. The characters in `,□+*-/()^` are considered special and the lexical value is equal to the character value.

Integer and floating values are recognized and evaluated to produce either T_Int or T_Float tokens. However, the decision is based on the value of *wantedType*; if it is T_Float, all numeric values are returned as T_Float; if it is T_Int, all numeric values are returned as T_Int.

produce T_Int and T_Float tokens. Names that take the form of a C or C++ identifier are assumed to be names of states defined at a higher level (states belonging to the parent galaxy or some ancestor galaxy). They are searched for using `lookup`; if not found, an error is reported using parseError and an error token is returned. If a State is found, a token of type T_ID is returned if it is an array state or COMPLEX; otherwise the state's current value is substituted and reparsed as a token. This means, for example, that a name of an IntState will be replaced with a T_Int token with the correct value.

```
const State* lookup(const char* name, Block* b);
```

This method searches for a state named *name* in Block *b* or one of its ancestors, and either returns it or a null pointer if not found.

```
void parseError (const char* part1, const char* part2 = "");
```

This method produces an appropriately formatted error message with the name of the state and the arguments and calls Error::abortRun.

```
static ParseToken pushback;
```

This is the pushback token, for use in parsing. Notice that it is static, which means that the state parser is not reentrant.

```
ParseToken evalIntExpression(Tokenizer& lexer);
ParseToken evalIntTerm(Tokenizer& lexer);
ParseToken evalIntFactor(Tokenizer& lexer);
ParseToken evalIntAtom(Tokenizer& lexer);
```

These four functions implement a simple recursive-descent expression parser.

An expression is either a term or a series of terms with intervening '+' or '-' signs.

A term is either a factor or a series of factors with interventing '*' or '/' signs.

A factor is either an atom or a series of atoms with intervening '^' signs for exponentiation. (Note, C fans! ^ means exponentiation, not exclusive-or!).

An atom is any number of optional unary minus signs, followed either by a parenthesized expression or a T_Int token.

If any of these methods reads too far, the pushback token is used. All getParseToken calls use *wantedType* T_Int, so any floating values in the expression are truncated to integer.

The token types returned from each of these methods will be one of T_Int, T_EOF, or T_ERROR.

```
ParseToken evalFloatExpression(Tokenizer& lexer);
ParseToken evalFloatTerm(Tokenizer& lexer);
ParseToken evalFloatFactor(Tokenizer& lexer);
ParseToken evalFloatAtom(Tokenizer& lexer);
```

These functions have the identical structure as the corresponding Int functions.

The token types returned from each of these methods will be one of T_Float, T_EOF, or T_ERROR.

## 8.2 types of states

### 8.2.1 Class IntState and class FloatState

Class IntState, derived from State, has an integer current value. Its `initialize()` function uses the `evalIntExpression` function to read an integer expression from the initial value string. If successful, it attempts to read another token from the string; if there is another token, it reports the error "extra text after valid expression".

An assignment operator is provided that accepts an integer value and loads it into the current value. A cast to integer is also defined for accessing the current value. The virtual function `currentValue` is overloaded to return a printed version of the current value.

In addition to the `setInitValue` from class State, a second form is provided that takes an integer argument.

Standard overrides for `isA`, `className`, and `clone` are provided.

Class FloatState is almost identical to class IntState except that its data field is a double precision value; where IntState functions have an argument or return value of `int`, FloatState has a corresponding argument or return value of `double`. Both are generated from the same pseudo-template files.

The `type()` function for IntState returns `"INT"`. For FloatState, `"FLOAT"` is returned. For both implementations, a prototype object is added to the KnownState list.

### 8.2.2 Class ComplexState

ComplexState is much like FloatState and IntState, except in the expressions it accepts for initial values. Its data member is Complex and it accordingly defines an assignment operator that takes a complex value and a conversion operator that returns one.

The initial value string for a ComplexState takes one of three forms: it may be the name of a galaxy ComplexState, a floating expression (of the form accepted by `State::evalFloatExpression`), or a string of the form

( *floatexp1* , *floatexp2* )

where both *floatexp1* and *floatexp2* are floating expressions. For the second form, the imaginary part will always be zero. For the third form, the first expression gives the real part and the second gives the imaginary part.

### 8.2.3 Class StringState

A StringState's current value is a string (more correctly, of type `const char*`). The current value is created by the `initialize()` function by scanning the initial value string. This string is copied literally, except that curly braces are special. If a pair of curly braces surrounds the name of a galaxy state, the printed representation of that state's current value (returned by the `currentValue` function) is substituted. To get a literal curly brace in the current value, prefix it with a backslash.

Class StringState defines assignment operators so that different string values can be copied to the current value; the value is copied with `savestring` (see Section 1.3 [utility functions], page 2) and deleted by the destructor.

### 8.2.4 numeric array states

Classes IntArrayState and FloatArrayState are produced from the same pseudo-template. Class ComplexArrayState has a very similar design. All return TRUE to `isArray`, provide an array element selection operator (`operator[](int)`), and an operator that converts the state into a pointer to the first element of its data (much like arrays in C).

The expression parser for FloatArrayState accepts a series of "subarray expressions", which are concatenated together to get the current value when `initialize()` is called. Subarray expressions may specify a single element, some number of copies of a single element, or a galaxy array state of the same type (another FloatArrayState). A single element specifier may either be a floating point value, a scalar (integer or floating) galaxy state name, or a general floating expression enclosed in parentheses. A number of copies of this single element can be specified by appending an integer expression enclosed in square brackets.

The expression parsers for IntArrayState and ComplexArrayState differ only that where FloatArrayState wants a floating expression, IntArrayState wants an integer expression and ComplexArrayState wants a complex expression (an expression suitable for initializing a ComplexState).

### 8.2.5 Class StringArrayState

As its name suggests, the current value for a StringArrayState is an array of strings. Whitespace in the initial value string separates "words"; each word is assigned by `initialize()` into a separate array element. Quotes can be use to permit "words" to have whitespace.

Current values of galaxy states can be converted into single elements of the StringArrayState value by surrounding their names with curly braces in the initial value. Galaxy StringArrayState names will be translated into a series of values.

There is currently no provision for modifying the current value of a StringArrayState other than calling of `initialize` to parse the current value string.

# 9 Support for known lists and such

Ptolemy is an extensible system, and in quite a few places it must create objects given only the name of that object. There are therefore several classes that are responsible for maintaining lists: the list of all known domains, of all known blocks, states, targets, etc. As a general rule, these classes support a `clone` or `makeNew` method to create a new object based on its name (you cannot clone a domain, however).

## 9.1 Class KnownBlock

The KnownBlock class is responsible for keeping a master list of all known types of Block objects in the system. All member functions of KnownBlock are static; the only non-static function of KnownBlock is the constructor.

The KnownBlock constructor has the form

```
KnownBlock(Block& block,const char* name);
```

The only reason for constructing a KnownBlock object is for the side effects; the side effect is to add *block* to the known block list for its domain under the name *name*, using `addEntry`.

The reason for using a constructor for this purpose is that constructors for global objects are called before execution of the main program; constructors therefore serve as a mechanism for execution of arbitrary initialization code for a module (as used here, "module" is an object file). Hence '`ptlang`', the Ptolemy star preprocessor, generates code like the following for star definitions:

```
static XXXMyClass proto;
static KnownBlock entry(proto,"MyClass");
```

This code adds a prototype entry of the class to the known list.

```
static void addEntry (Block &block, const char* name, int onHeap);
```

This function actually adds the block to the list. If *onHeap* is true, the block will be destroyed when the entry is removed or replaced from the list. Separate lists are maintained for each domain; the block is added to the list corresponding to *block*`.domain()`.

```
static const char* domain();
```

Return the current domain name.

```
static int setDomain (const char* newDom);
```

Change the current domain. Return TRUE if it worked, FALSE for an unknown domain. This affects which sublist of blocks is searched by subsequent `find` and `clone` calls.

```
static const Block* find (const char* name);
```

The find method returns a pointer the appropriate block in the current domain. A null pointer is returned if no match is found.

```
static Block* clone (const char* name);
static Block* makeNew (const char* name);
```

The `clone` method takes a string, finds the appropriate block in the current domain, and returns a clone of that block (the `clone` method is called on the block. This method, as a rule, generates a duplicate of the block.

The `makeNew` function is similar except that `makeNew` is called on the found block. As a rule, `makeNew` returns an object of the same class, but with default initializations (for example, with default state values).

For either of these, an error message is generated (with `Error::abortRun`) and a null pointer is returned if there is no match.

```
static StringList nameList();
static StringList nameList (const char* domain);
```

Return the names of known blocks in the current domain (first form) or the given domain (second form). Names are separated by newline characters.

```
static int isDynamic (const char* type);
```

Return true if the named block is dynamically linked.

There is an iterator associated with KnownBlock, called KnownBlockIter. It takes as an argument the name of a domain. The argument may be omitted, in which case the current domain is used. Its `next` function returns the type `const Block *`; it steps through the blocks on the known list for that domain.

## 9.2 Class KnownTarget

The KnownTarget class keeps track of targets in much the same way that KnownBlock keeps track of blocks. There are some differences: there is only a single list of targets, not one per domain as for blocks.

The constructor works exactly the same way that the constructor for KnownBlock works; the code

```
static MyTarget proto(args);
static KnownTarget entry(proto,"MyTarget");
```

adds the prototype instance to the known list with a call to addEntry.

```
static void addEntry (Target &target, const char* name, int onHeap);
```

This function actually adds the Target to the list. If *onHeap* is true, the target will be destroyed when the entry is removed or replaced from the list. There is only one list of Targets.

```
static const Target* find (const char* name);
```

The find method returns a pointer the appropriate target. A null pointer is returned if no match is found.

```
static Target* clone (const char* name);
```

The `clone` method takes a string, finds the appropriate target on the known target list, and returns a clone of that block (the `clone` method is called on the target. This method, as a rule, generates a duplicate of the target. An error message is generated (with `Error::abortRun`) and a null pointer is returned if there is no match.

```
static int getList (const Block& b, const char** names, int nMax);
```

This function returns a list of names of targets that are compatible with the Block *b*. The return value gives the number of matches. The *names* array can hold *nMax* strings; if there are more, only the first *nMax* are returned.

```
static int getList (const char* dom, const char** names, int nMax);
```

This function is the same as above, except that it returns names of targets that are compatible with stars of a particular domain.

```
static int isDynamic (const char* type);
```

Return true if there is a target on the known list named *type* that is dynamically linked; otherwise return false.

```
static const char* defaultName(const char* dom = 0);
```

Return the default target name for a domain (default: current domain).

There is an iterator associated with KnownTarget, called KnownTargetIter. Since there is only one known target list, it is unusual for an iterator in that it takes no argument for its constructor. Its `next` function returns the type `const Target *`; it steps through the targets on the known list.

## 9.3 Class Domain

The Domain class represents the information that Ptolemy needs to know about a particular domain so that it can create galaxies, wormholes, nodes, event horizons, and such for that domain. For each domain, the designer creates a derived class of Domain and one prototype object. Thus the Domain class has two main parts: a static interface, which manages access to the list of Domain objects, and a set of virtual functions, which provides the standard interface for each domain to describe its requirements.

### 9.3.1 Domain virtual functions

```
virtual Star& newWorm(Galaxy& innerGal,Target* innerTarget = 0);
```

This function creates a new wormhole with the given inner galaxy and inner target. The default implementation returns an error.

XXXDomain might override this as follows:

```
Star& XXXDomain::newWorm(Galaxy& innerGal,Target* innerTarget)  {
        LOG_NEW; return *new XXXWormhole(innerGal,innerTarget);
}
```

```
virtual EventHorizon& newFrom();
virtual EventHorizon& newTo();
```

These functions create event horizon objects to represent the XXXfromUniversal and XXXtoUniversal functions. The default implementations return an error.

XXXDomain might override these as

```
EventHorizon& XXXDomain::newFrom() {
    LOG_NEW; return *new XXXfromUniversal;
}

EventHorizon& XXXDomain::newTo() {
    LOG_NEW; return *new XXXtoUniversal;
}
```

```
virtual Geodesic& newNode() = 0;
```

This function creates a new permanent node appropriate for netlist connections for the domain. There is no default implementation.

```
virtual int isGalWorm();
```

This function returns FALSE by default. If overridden by a function that returns TRUE, a wormhole will be created around every galaxy for this domain.

## 9.4 Class KnownState

KnownState manages two lists of states, one to represent the types of states known to the system (integer, string, complex, array of floating, etc), and one to represent certain predeclared global states.

It is very much like KnownBlock (see Section 9.1 [class KnownBlock], page 94) in internal structure. Since it manages two lists, there are two kinds of constructors.

`KnownState (State` *`&state`*`, const char*` *`name`*`);`

This constructor adds an entry to the state type list. For example,

```
static IntState proto;
static KnownState entry(proto,"INT");
```

permits IntStates to be produced by cloning. The *type* argument must be in uppercase, because of the way `find` works (see below).

The second type of constructor takes three arguments:

```
KnownState (State &state, const char* name, const char* value);
```

This constructor permits names to be added to the global state symbol list, for use in state expresssions. For example, we have

```
static FloatState pi;
KnownState k_pi(pi,"PI","3.14159265358979323846");
```

```
static const State* find (const char* type);
```

The `find` method returns a pointer the appropriate prototype state in the state type list. The argument is always changed to uppercase. A null pointer is returned if there is no match.

```
static const State* lookup (const char* name);
```

The lookup method returns a pointer to the appropriate state in the global state list, or null if there is no match.

```
static State* clone (const char* type);
```

The clone method takes a string, finds the appropriate state using `find`, and returns a clone of that block. A null pointer is returned if there is no match, and `Error::error` is also called.

```
static StringList nameList();
```

Return the names of all the known state types, separated by newlines.

```
static int nKnown();
```

Return the number of known states.

# 10 I/O classes

## 10.1 StringList, a kind of String class

Class StringList provides a subset of the functions provided by the typical C++ String class; enough for our purposes. It is privately derived from SequentialList (see Section 1.5 [class SequentialList], page 4).

Its internal implementation is as a list of `char *` strings, each on the heap. A StringList object can be treated either as a single string or as a list of strings; the individual substrings retain their separate identity until the conversion operator to type `const char *` is invoked. There are also operators that add numeric values to the StringList; there is only one format available for such additions.

### 10.1.1 StringList constructors and assignment operators

The default constructor makes an empty StringList. There is also a copy constructor and five single-argument constructors that can function as conversions from other types to type StringList; they take arguments of the types `char`, `const char *`, `int`, `double`, and `unsigned int`.

There are also six assignment operators corresponding to these constructors: one that takes a `const StringList&` argument and also one for each of the five standard types: `char`, `const char *`, `int`, `double`, and `unsigned int`.

The resulting object has one piece, unless initialized from another StringList in which case it has the same number of pieces.

### 10.1.2 adding to StringLists

There are six functions that can add a printed representation of an argument to a StringList: one each for arguments of type `const StringList&`, `char`, `const char *`, `int`, `double`, and `unsigned int`. In each case, the function can be accessed in either of two equivalent ways:

```
StringList& operator += (type arg);
```

```
StringList& operator << (type arg);
```

The second "stream form" is considered preferable; the "+=" form is there for backward compatibility. If a StringList object is added, each piece of the added StringList is added separately (boundaries between pieces are preserved); for the other five forms, a single piece is added.

### 10.1.3 StringList information functions

```
const char* head() const;
```

Return the first substring on the list (the first "piece"). A null pointer is returned if there are none.

```
int length() const;
```

Return the length in characters.

```
int numPieces() const;
```

Return the number of substrings in the StringList.

### 10.1.4 StringList conversion to const char *

```
operator const char* ();
```

This function joins all the substrings in the StringList into a single piece, so that afterwards `numPieces` will return 1. A null pointer is always returned if there are no characters.

Warning: if this function is called on a temporary StringList, it is possible that the compiler will delete the StringList object before the last use of the returned `const char *` pointer. The result is that the pointer may wind up pointing to garbage. The best workaround for such problems is to make sure that any StringList object "has a name" before this conversion is applied to it; e.g. assign the results of functions returning StringList objects to local StringList variables or references before trying to convert them.

```
char* newCopy() const;
```

This function makes a copy of the StringList's text in a single piece as a `char *` in dynamic memory. The object itself is not modified. The caller is responsible for deletion of the returned text.

### 10.1.5 StringList destruction and zeroing

```
void initialize();
```

This function deallocates all pieces of the StringList and changes it to an empty StringList.

```
~StringList();
```

The destructor calls the `initialize` function.

### 10.1.6 Class StringListIter

Class StringListIter is a standard iterator that operates on StringLists. Its `next()` function returns a pointer of type `const char *` to the next substring of the StringList. It is important to know that the operation of converting a StringList to a `const char *` string joins all the substrings into a single string, so that operation should be avoided if extensive use of StringListIter is planned.

## 10.2 Tokenizer, a simple lexical analyzer class

The Tokenizer class is designed to accept input for a string or file and break it up into tokens. It is similar to the standard istream class in this regard, but it has some additional facilities. It permits character classes to be defined to specify that certain characters are whitespace and others are "special" and should be returned as single-character tokens; it permits quoted strings to override this, and it has a file inclusion facility. In short, it is a simple, reconfigurable lexical analyzer.

Tokenizer has a public const data member named `defWhite` that contains the default whitespace characters: space, newline, and tab. It is possible to change the definition of whitespace for a particular constructor.

### 10.2.1 Initializing Tokenizer objects

Tokenizer provides three different constructors:

```
Tokenizer();
```

The default constructor creates a Tokenizer that reads from the standard input stream, `cin`. Its special characters are simply `(` and `)`.

```
Tokenizer(istream& input,const char* spec,
          const char* w = defWhite);
```

This constructor creates a Tokenizer that reads from the stream named by *input*. The other arguments specify the special characters and the whitespace characters.

```
Tokenizer(const char* buffer,const char* spec,
          const char* w = defWhite);
```

This constructor creates a Tokenizer that reads from the null-terminated string in *buffer*.

Tokenizer's destructor closes any include files associated with the constructor and deletes associated internal storage.

The following operations change the definition of whitespace and of special characters, respectively:

```
const char* setWhite(const char* w);
const char* setSpecial(const char* s);
```

In each case, the old value is returned.

By default, the line comment character for Tokenizer is `#`. It can be changed by

```
char setCommentChar(char n);
```

Use an argument of 0 to disable the feature. The old comment character is returned.

### 10.2.2 Reading from Tokenizers

The next operation is the basic mechanism for reading tokens from the Tokenizer:

```
Tokenizer& operator >> (char * pBuffer);
```

Here *pBuffer* points to a character buffer that reads the token. There is a design flaw: there isn't a way to give a maximum buffer length, so overflow is a risk.

By analogy with streams, the following operation is provided:

```
operator void* ();
```

It returns null if EOF has already been reached and non-null otherwise. This permits loops like

```
Tokenizer tin;
while (tin) { ... do stuff ... }
```

```
int eof() const;
```

Returns true if the end of file or end of input has been reached on the Tokenizer. It is possible that there is nothing left in the input but write space, so in many situations `skipwhite` should be called before making this test.

```
void skipwhite();
```

Skip whitespace in the input.

```
void flush();
```

If in an include file, the file is closed. If at the top level, discard the rest of the current line.

### 10.2.3 Tokenizer include files

Tokenizer can use include files, and can nest them to any depth. It maintains a stack of include files, and as EOF is reached in each file, it is closed and popped off of the stack.

The method

```
    int fromFile(const char* name);
```

opens a new file and the Tokenizer will then read from that. When that file ends, Tokenizer will continue reading from the current point in the current file.

```
    const char* current_file() const;
    int current_line() const;
```

These methods report on the file name and line number where Tokenizer is currently reading from. This information is maintained for include files. At the top level, `current_file` returns a null pointer, but `current_line` returns one more than the number of line feeds seen so far.

```
    int readingFromFile() const;
```

Returns true (1) if the Tokenizer is reading from an include file, false (0) if not.

## 10.3 pt_ifstream and pt_ofstream: augmented fstream classes

The classes pt_ifstream and pt_ofstream are derived from the standard stream classes ifstream and ofstream, respectively. They are defined in the header file '`pt_fstream.h`'. They add the following features:

First, certain special "filenames" are recognized. If the filename used in the constructor or an open call is `<cin>`, `<cout>`, `<cerr>`, or `<clog>` (the angle brackets must be part of the string), then the corresponding standard stream of the same name is used for input (pt_ifstream) or output (pt_ofstream). In addition, C standard I/O fans can specify `<stdin>`, `<stdout>`, or `<stderr>` as well.

Second, the Ptolemy `expandPathName` is applied to the filename before it is opened, permitting it to start with `~user` or `$VAR`.

Finally, if a failure occurs when the file is opened, `Error::abortRun` is called with an appropriate error message, including the Unix error condition.

Otherwise these classes are identical to the standard ifstream and ofstream classes and can be used as replacements.

## 10.4 XGraph, an interface to the xgraph program

The XGraph class provides an interface for the '`xgraph`' program for plotting data on an X window system display. The modified '`xgraph`' program provided with the Ptolemy distribution should be used, not the contributed version from the X11R5 tape.

The constructor for XGraph does not completely initialize the object; initialization is completed by the `initialize()` method:

```
void initialize(Block* parent, int noGraphs,
     const char* options, const char* title,
     const char* saveFile = 0, int ignore = 0);
```

The *parent* argument is the name of a Block that is associated with the XGraph object; this Block is used in `Error::abortRun` messages to report errors.

*noGraphs* specifies the number of data sets that the graph will contain. Each data set is a separate stream and is plotted in a different color (a different line style for B/W displays).

*options* is a series of command line options that will be passed unmodified to the xgraph program. It is subject to expansion by the Unix shell.

*title* is the title for the graph; it can contain special characters (it is *not* subjected to expansion by the Unix shell).

*saveFile* is the name of a file to save the graph data into, in ASCII form. If it is not given, the data are not saved, and a faster binary format is used.

*ignore* specifies the number of initial points to ignore from each data set.

```
void setIgnore(int n);
```

Reset the "ignore" parameter to *n*.

```
void addPoint(float y);
```

Add a single point to the first data set whose X value is automatically generated (0, 1, 2, 3... on successive calls) and whose Y value is *y*.

```
void addPoint(float x, float y);
```

Add the point (*x*, *y*) to the first data set.

```
void addPoint(int dataSet, float x, float y);
```

Add the point (*x*, *y*) to the data set indicated by *dataSet*. Data sets start with 1.

```
void newTrace(int dataSet = 1);
```

Start a new trace for the nth dataset. This means that there will be no connecting line between the last point plotted and the next point plotted.

```
void terminate();
```

This function flushes the data out to disk, closes the files, and invokes the xgraph program.

If the destructor is called before `terminate`, it will close and delete the temporary files.

## 10.5 Histogram classes

The Historgram class accepts a stream of data and accumulates a histogram. The XHistogram class uses a Histogram to collect the data and an XGraph to display it.

### 10.5.1 Class Histogram

The Histogram class accumulates data in a histogram. Its constructor is as follows:

```
Histogram(double width = 1.0, int maxBins = HISTO_MAX_BINS);
```

The default maximum number of bins is 1000. The bin centers will be at integer multiples of the specified bin width. The total width of the histogram depends on the data; however, there will always be a bin that includes the first point.

```
void add(double x);
```

Add the point $x$ to the histogram.

```
int numCounts() const;
double mean() const;
double variance() const;
```

Return the number of counts, the mean, and the variance of the data in the histogram.

```
int getData(int binno, int& count, double& binCenter);
```

Get counts and bin centers by bin number, where 0 indicates the smallest bin. Return TRUE if this is a valid bin. Thus the entire histogram data can be retrieved by stepping from 0 to the first failure.

## 10.5.2 Class XHistogram

An XHistogram object has a private XGraph member and a private Histogram member. The functions

```
int numCounts() const;
double mean() const;
double variance();
```

simply pass through to the Histogram object, and

```
void addPoint(float y);
```

adds a point to the histogram and does other bookkeeping. There are two remaining methods:

```
void initialize(Block* parent, double binWidth,
   const char* options, const char* title,
   const char* saveFile, int maxBins = HISTO_MAX_BINS);
```

This method initializes the graph and histogram object. *parent* is the parent Block, used for error messages. *binWidth* and *maxBins* initialize the Histogram object. *options* is a string that is included in the command line to the xgraph program; other options, including `-bar -nl -brw` *value*, are passed as well. *title* is the graph title, and *saveFile*, if non-null, gives a file in which the histogram data is saved (this data is the histogram counts, not the data that was input with `addPoint`).

```
void terminate();
```

This method completes the histogram, flushes out the temporary files, and executes xgraph.

# 11 Miscellaneous classes

This section includes classes that did not fit elsewhere.

## 11.1 Mathematical classes

### 11.1.1 class Complex

Class Complex is a simple subset of functions provided in the Gnu and AT&T complex classes. The standard arithmetic operators are implemented, as are the assignment arithmetic operators `+=`, `-=`, `*=`, and `/=`, and equality and inequality operators `==` and `!=`. There is also `real()` and `imag()` methods for accessing real and imaginary parts.

It was originally written when libg++ was subject to the GPL. The current licensing for libg++ does not prevent us from using it and still distributing Ptolemy the way we want, but having it makes ports to other compilers (e.g. cfront) easier.

The following non-member functions take Complex arguments:

```
Complex conj(const Complex& arg);
double real(const Complex& arg);
double imag(const Complex& arg);
double abs(const Complex& arg);
```

Return the conjugate, real part, imaginary part, or absolute value, respectively.

```
double arg(const Complex& arg);
```

Return the angle between the X axis and the vector made by the argument. The expression

```
abs(z)*exp(Complex(0.,1.)*arg(z))
```

is in theory always equal to z.

```
double norm(const Complex& arg);
```

return the absolute value squared.

```
Complex sin(const Complex& arg);
Complex cos(const Complex& arg);
Complex exp(const Complex& arg);
Complex log(const Complex& arg);
Complex sqrt(const Complex& arg);
```

Standard mathematical functions. `log` returns the principal logarithm.

```
Complex pow(double base,const Complex& expon);
Complex pow(const Complex& base, const Complex& expon);
```

Raise base to expon power.

There is also an << operator to print a Complex on an ostream.

## 11.1.2 class Fraction

Class Fraction represents fractions. The header '`Fraction.h`' also provides declarations for the `lcm` (least common multiple) and `gcd` (greatest common divisor) functions, as these functions are needed for Fraction but are generally useful.

```
Fraction ();
Fraction (int num, int den=1);
```

The default constructor produces a fraction with numerator 0 and denominator 1. The other constructor allows the numerator and denominator to be specified arbitrarily.

```
int num() const;
int den() const;
```

Return the numerator or denominator.

```
operator double() const;
```

Return the value of the fraction as a double.

Class Fraction implements the basic binary math operators `+`, `-`, `*`, `/`; the assignment operators `=`, `+=`, `-=`, `*=`, and `/=`, and the equality test operators `==` and `!=`.

The method

```
Fraction& simplify();
```

reduces the fraction to lowest terms, and returns a reference to the fraction.

There is also an `<<` operator to print a Fraction on an ostream.

## 11.2 Class IntervalList

The IntervalList class represents a set of unsigned integers, represented as a series of intervals of integers that belong to the set.

It is built on top of a class Interval that represents a single interval.

There is also a text representation for IntervalLists. This representation can be used to read or write IntervalList objects to streams, and also can be used in the IntervalList constructor. This text representation looks exactly like the format the "rn" newsreader uses to record which articles have been read in a Usenet newsgroup (which is where we got it from; thank you, Larry Wall).

In the text representation, an IntervalList is specified as one or more Intervals, separated by commas. An Interval is either an unsigned integer or two unsigned intervals with an intervening minus sign. Here is one possible IntervalList specification:

1-1003,1006,1008-1030,1050

Whitespace is not permitted in this representation.

IntervalList specifiers do not have to be in increasing order, but if they are not, they are changed to "canonical form", in which any overlapping intervals are merged and the intervals are sorted to appear in increasing order.

An IntervalList is best thought of as a set of unsigned integers. Its methods in many cases perform set operations: forming the union, intersection, or set difference of two IntervalLists.

### 11.2.1 class Interval and methods

The Interval class is in some ways simply an implementation detail of class IntervalList, but since its existence is exposed by public methods, it is documented here.

An Interval has an *origin* and a *length*, and represents the set of *length* unsigned integers beginning with *origin*. It also has a pointer that can point to another Interval.

The constructor

```
Interval(unsigned origin=0, unsigned length=0,
    Interval* nxt = 0);
```

permits all these members to be set. The copy constructor copies the origin and length values but always sets the next pointer to null. A third constructor

```
Interval(const Interval& i1,Interval* nxt);
```

permits a combination of a copy and a next-pointer initialization.

The members

```
unsigned origin() const;
unsigned length() const;
```

return the origin and length values.

```
unsigned end() const;
```

The end function returns the last unsigned integer that is a member of the Interval; 0 is returned for empty Intervals.

There are a number of queries that are valuable for building a set class out of Intervals:

```
int isAfter(const Interval &i1) const;
```

`isAfter` returns true if this Interval begins after the end of interval *i1* .

```
int endsBefore(const Interval &i1) const;
```

`endsBefore` returns true if this Interval ends strictly before the origin of interval i1.

```
int completelyBefore(const Interval &i1) const;
```

`completelyBefore` returns true if `endsBefore` is true and there is space between the intervals (they cannot be merged).

```
int mergeableWith(const Interval& i1) const;
```

`mergeableWith` returns true if two intervals overlap or are adjacent, so that their union is also an interval.

```
int intersects(const Interval& i1) const;
```

`intersects` returns true if two intervals have a non-null intersection.

```
int subsetOf(const Interval& i1) const;
```

`subsetOf` returns true if the argument is a subset of this interval.

```
void merge(const Interval& i1);
```

`merge` alters the interval to the result of merging it with *i1*. It is assumed that `mergeableWith` is true.

```
Interval& operator&=(const Interval& i1);
```

This Interval is changed to the intersection of itself and of *i1*.

### 11.2.2 IntervalList public members

```
IntervalList();
```

The default constructor produces the empty IntervalList.

```
IntervalList(unsigned origin,unsigned length);
```

This constructor creates an IntervalList containing *length* integers starting with *origin*.

```
IntervalList(const char* definition);
```

This constructor takes a definition of the IntervalList from the string in *definition*, parses it, and creates the list of intervals accordingly.

There is also a copy constructor, an assignment operator, and a destructor.

```
int contains(const Interval& i1) const;
```

The `contains` method returns 0 if no part of *i1* is in the IntervalList, 1 if *i1* is completely contained in the IntervalList, and -1 if *i1* is partially contained (has a non-null intersection).

```
IntervalList& operator|=(const Interval& src);
IntervalList& operator|=(const IntervalList& src);
```

The `|=` operators sets the IntervalList to the union of itself and the Interval, or the IntervalList, *src*.

```
IntervalList operator&(const IntervalList& arg) const;
```

The binary & operator returns the intersection of its arguments, which are not changed.

```
IntervalList& subtract(const Interval& i1);
IntervalList& operator-=(const Interval& i1);
```

Subtract the Interval *il* from the list. That is, any intersection is removed from the set. Both the `subtract` and `-=` forms are equivalent.

```
IntervalList& operator-=(const IntervalList &arg);
```

This one subtracts the argument *arg* from the list (removes any intersection).

```
int empty() const;
```

Return TRUE (1) for an empty IntervalList, otherwise FALSE (0).

### 11.2.3 IntervalList iterator classes.

There are two iterator classes associated with IntervalList, IntervalListIter and CIntervalListIter. The only difference is that the latter iterator can be used with const IntervalList objects and returns pointers to const Interval objects; the former requires a non-const IntervalIList and returns pointers to Interval.

These objects obey the standard iterator interface; the `next()` or `++` function returns a pointer to the next contained Interval; `reset` goes back to the beginning.

## 11.3 Classes for interacting with the system clock

These classes provide simple means of interacting with the operating system's clock – sleeping until a specified time, timing events, etc. They may be replaced with something more general.

class TimeVal represents a time interval to microsecond precision. There are three constructors:

```
TimeVal();
TimeVal(long sec, long usec);
TimeVal(double length);
```

The first represents a time interval of zero. The second represents a time interval with the specified number of seconds and microseconds. In the third case, the *length* argument is rounded to the nearest multiple of one millionth.

These classes rely on features found in BSD-based Unix systems and newer System V Unix systems. Older System V systems tend not to provide the ability to sleep for a time specified more accurately than a second.

```
operator double() const;
```

This returns the interval value as a double.

```
TimeVal operator +(const TimeVal& arg) const;
TimeVal operator -(const TimeVal& arg) const;
TimeVal& operator +=(const TimeVal& arg);
TimeVal& operator -=(const TimeVal& arg);
```

These operators do simple addition and subtraction of TimeVals.

class Clock provides a simple interface to the system clock for measurement of actual elapsed time. It has an internal TimeVal field that represents the starting time of a time interval.

```
Clock();
```

The constructor creates a Clock with starting time equal to the time at which the constructor is executed.

```
void reset();
```

This method resets the start time to "now".

```
TimeVal elapsedTime() const;
```

This method returns the elapsed time since the last `reset` or the call to the constructor.

```
int sleepUntil(const TimeVal& howLong) const;
```

This method causes the process to sleep until *howLong* after the start time.