Chapter 6. Classes for connections between blocks

Authors: *Joseph T. Buck*

Other Contributors: *Tom Lane*
Yuhong Xiong

This chapter describes the classes that implement connections between blocks. For simulation domains, these classes are responsible for moving objects called Particles 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. Currently, class PortHole is also responsible for the type resolution algorithm that assigns specific types to `ANYTYPE` portholes. It would probably be better to put that function in Geodesic, which would make it simpler to provide domain-specific type resolution rules. This improvement must await a redesign of the PortHole/Geodesic structure.

6.1 Class GenericPort

The class GenericPort is a base class that provides common elements between class PortHole and class MultiPortHole. 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 . 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. The `type()` and `typePort()` functions belong to GenericPort, not PortHole, because multiportholes have a declared type and can be type-equivalenced to other portholes. However, type resolution is strictly a PortHole notion. Multiportholes need no resolved type because they do not themselves transport particles, and indeed the concept would be ambiguous since the member ports of a multiporthole might have different resolved types. The declared type of a multiporthole is automatically assigned to its children, and its children are automatically brought into any type equivalence set the multiporthole is made part of. Thereafter, type resolution considers only the member portholes and not the multiporthole itself.

6.1.1 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 GenericPort.

`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 must be resolved during setup. Note that `type()` returns the port’s declared type, as supplied to `setPort()`. This is not necessarily the datatype that will be chosen to pass through the port at runtime. That type is available from the `PortHole::resolvedType()` function.

`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() << "\back 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.

6.1.2 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 data type.

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

Link to another port for determining the type of ‘`ANYTYPE`’ connections. The "inheritance" relationship is actually a completely symmetric constraint, and so this function would have been better named `sameTypeAs()`. Any number of portholes can be tied together by `inheritTypeFrom()` calls. Internally this is represented by chaining all the members of such a type equivalence set into a circular loop, which can be walked via `typePort()` calls. If a multiporthole is made part of a type equivalence set, all its current and future children become part of the set automatically.

```
virtual void connect(GenericPort& destination,int numberDelays,
                   const char* initDelayValues = 0);
```

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.

6.1.3 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, `alias()->translateAliases()` is returned.

6.2 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.

6.2.1 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 base class 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.

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

Return the data type 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`. Two connected portholes will always return the same resolvedType. A null pointer will be returned if the type has not yet been resolved, e.g. before initialization.

`DataType preferredType () const;`

Return the "preferred" type of the porthole. This is the same as the declared type (`GenericPort::type()`) if the declared type is not `ANYTYPE`. If the declared type is `ANYTYPE`, the preferredType is the type of the connected porthole or type equivalence set from which the `ANYTYPE`'s true type was determined. (If preferredType and resolvedType are not the same, the need for a run-time type conversion is indicated. Code generation domains may choose to splice in type conversion stars to ensure that preferredType and resolvedType are the same at all ports.) A null pointer will be returned if the type has not yet been resolved, 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.

`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.

`virtual Geodesic* allocateGeodesic();`

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

`void enableLocking(const PtGate& master);`

Enable locking on access to the Geodesic and Plasma. This is appropriate for connections that cross thread boundaries. Assumption: `initialize()` has been called.

`void disableLocking();`

The converse.

`int isLockEnabled() const;`
Returns the lock status.

6.2.2 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.

`int numberTokens;`
Number of Particles stored in the buffer each time the Geodesic is accessed. Normally this is one except for dataflow-type stars, where it is the number of Particles consumed or generated.

`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.

`virtual int allocatePlasma();`
Allocate Plasma (default method uses global Plasma).

`int allocateLocalPlasma();`
Alternate function allocates a local Plasma (for use in derived classes).

`void deletePlasma();`
Delete Plasma if local; detach other end of connection from Plasma as well.

`void allocateBuffer();`

Allocate new buffer.

`DataType SetPreferredType();`

Function to determine preferred types during initialization. Returns the preferred type of this porthole, or 0 on failure. Protected, not private, so that subclasses that override `set-ResolvedType()` can call it.

6.2.3 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.

`void initialize();`

Zero out the contents 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).

6.3 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.

6.3.1 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.

6.3.2 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 .

`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.

6.4 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.

6.4.1 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).

6.4.2 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).

6.5 Class ParticleStack

ParticleStack is an efficient base class 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).

`Particle* head() const;`

Return pointer to head.

`Particle* tail() const;`

Return pointer to tail.

6.6 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 . The base class 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 "heavyweight" 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). A severe limitation of the current Geodesic class is that it is designed around point-to-point connections, ie, a single source port to a single destination port. This is a problem for domains that wish to support one-to-many geodesics (single source to multiple receivers) or many-to-many geodesics (such as multiple in/out ports connected to a common bus). Geodesic ought to be redesigned as a base class that supports any number of connected ports, with the restriction to point-to-point being a specialized subclass. This would also allow a cleaner treatment of autofork (autoforking geodesics could just be a subclass of Geodesic). It would be necessary to remove PortHole's belief that there is a unique far-side porthole, and that would require rethinking the porthole type resolution algorithm; probably type resolution should become a Geodesic function, not a PortHole function. This area will be addressed in some future version of Ptolemy.

6.6.1 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 within 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 &`*`dp`*`);`

Set the destination port to `dp.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 = 0) 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.

`int maxNumParticles() const;`

Return maximum number of particles.

`virtual void makeLock(const PtGate& master);`

Create a lock for the Geodesic.

`virtual void delLock();`

Delete lock for the Geodesic.

```
int isLockEnabled() const;
```
Return lock status.

```
const char * initDelayValues();
```
Return the *initValues* string.

6.6.2 Geodesic protected members

```
void portHoleConnect();
```
This function completes a connection if the originating and destination ports are set up.

```
virtual Particle* slowGet();
virtual void slowPut(Particle*);
```
The “slow” versions of `get()` and `put()`.

```
PortHole *originatingPort;
PortHole *destinationPort;
```
These protected members point to my neighbors.

6.7 Class Plasma

Class Plasma is a pool for particles. It is derived from ParticleStack . 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).

```
int isLocal() const;
```
Returns *localFlag*.

```
static Plasma* getPlasma (DataType t);
```
Get the appropriate global Plasma object given a type.

```
static Plasma* makeNew (DataType t);
```
Create a local Plasma object given a type.

```
void makeLock(const PtGate& master);
```
Create a lock for the Plasma.

`void delLock();`

Delete lock for the Plasma. No effect on global plasmas.

`short incCount();`

Increase reference count, when adding reference from PortHole to a local Plasma. New count is returned. Global Plasmas pretend their count is always 1.

`short decCount();`

Decrease reference count, when removing reference from PortHole to a local Plasma. New count is returned. Idea is we can delete it if it drops to zero. Global Plasmas pretend their count is always 1.

`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.

6.8 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)`.

6.9 Classes for Galaxy ports

Class GalPort is derived from class PortHole . Class GalMultiPort is derived from class MultiPortHole . These classes are used by InterpGalaxy , 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.

6.10 The PortHole type resolution algorithm

The type resolution algorithm is concerned with assigning concrete types to `ANYTYPE` portholes and resolving conflicts between the types of connected portholes. The algorithm is somewhat complex since it tries to produce convenient results in an area where the "right" behavior is not always easy to define. Ptolemy 0.7 introduces a new resolution algorithm that is hoped to produce more convenient and less surprising results than the method previously used. The problem of connecting ports of different types is simple to resolve: we say that the input porthole determines what particle type to use, and that any necessary type conversion takes place when the output porthole puts data into a particle (or buffer variable, in the case of codegen domains). The opposite convention would be about equally defensible, but this is the one that has historically been used in Ptolemy. It has the advantage that star writers can presume that the declared type of an input porthole is the data type that will actually be received, whenever the declared type is not `ANYTYPE`. Resolving `ANYTYPE` portholes is much more difficult. We need to handle several fundamental cases:

1. Printer and similar polymorphic stars, which accept any input type. They simply declare their inputs to be `ANYTYPE`, and we need to resolve them to the type of the connected output. (Introducing any forced particle type conversion would be very

undesirable.)

2. Fork and similar stars, which want to bind multiple outputs to the type of a given input. Here the input porthole and output portholes are `ANYTYPE`, and are bound into a type equivalence set by `inheritTypeFrom()`. If possible we want to resolve all these portholes to the type of the output porthole connected to the input.

3. Merge and similar stars, which have a single output type-equivalenced to multiple inputs. If the inputs all receive the same type of data, we should resolve the output to that type. If there is no common input type, but the input connected to the single output has determinable type, we can resolve the output porthole to that type (since the data would ultimately get converted to that type, anyway). If the connected input is `ANYTYPE`, we must declare error, because we have no good way to choose a type for the Merge's output.

We have to recursively propagate type information in order to deal with chains of `ANYTYPE` stars, such as one Fork following another. In some cases the type is really undefined. Consider this universe (using ptcl syntax):

```
star f Fork; star p Printer
connect f output f input 1
connect f output p input
```

There are no types anywhere in the system. We have little choice but to declare an error. Thus, the fact that we will sometimes fail to assign a type is not an implementation shortcoming but an unavoidable property of the problem. These considerations lead to the following algorithm. We first perform a recursive scan to resolve `ANYTYPE` portholes, the results of which are represented by a "preferred" type assigned to each porthole. (A porthole of non-`ANYTYPE` declared type always has that type as its preferred type; so preferred type assignment is only interesting for `ANYTYPE` portholes.) Then, the "resolved" type of each connected pair of input and output portholes is the preferred type of the input porthole. This is the type that will be used for actual data transported between those portholes. It is useful to explicitly store the preferred type, so that codegen domains can detect type mismatches just by looking at individual output portholes. A type conversion star can be spliced in wherever an output is found that has `preferredType != resolvedType`. Assignment of preferred types proceeds in two passes. Pass 1 is "feed forward" from outputs to inputs. Pass 2 is "feed back" from inputs to outputs; it is the dual of Pass 1. Pass 2 is invoked only if Pass 1 is unable to choose a type for a porthole. The details are:

Pass 1:

Non-`ANYTYPE` portholes are simply assigned their declared type as preferred type. If an `ANYTYPE` porthole is not a member of an equivalence group, but is an input porthole and is connected to a porthole of pass-1-assignable type, that porthole's type becomes its preferred type. When an `ANYTYPE` porthole is a member of an equivalence group, the group is scanned to see if it includes any non-`ANYTYPE` portholes; if so, they must all agree in type, and that type becomes the preferred type of all members of the group. But usually, all the members of an equivalence set will be `ANYTYPE`. Then, pass 1 scans all the input portholes of the group to see whether their connected portholes have pass-1-assignable type. If at least one does, and all of the assignable ones have the same preferred type, then that common type becomes the preferred type of all the members of the equivalence group.

Pass 2:

If an unassigned `ANYTYPE` porthole is not a member of an equivalence group, but is connected to a porthole of type assignable by either pass 1 or pass 2, that porthole's type becomes its preferred type. When an `ANYTYPE` porthole is a member of an equivalence group, all the output portholes of the group are scanned to see whether their connected portholes have type assignable by either pass 1 or pass 2. If at least one does, and all of the assignable ones have the same preferred type, then that common type becomes the preferred type of all the members of the equivalence group.

Pass 1 handles Fork-like stars as well as Merge stars whose inputs all have the same type. Pass 2 does something reasonable with Merge stars that have inputs of different types: if the merge output is going to a port of knowable type, we may as well just output particles of that type. An error is declared if a porthole's type remains unassigned after both passes. This occurs if a Merge-like star has inputs of nonidentical types and an output connected to an (unresolvable) `ANYTYPE` input. The user must insert type conversion stars to coerce the Merge inputs to a common type, so that the system can figure out what type to use for the Merge output. Notice that each pass will resolve an equivalence set if all the **assignable** connected portholes agree on a type; it is not required that all the connected portholes be assignable. This rule is needed to allow resolution of schematics that contain type-free loops. Here is an example:

```
star imp Impulse; star f Fork; star d Delay; star p Printer
connect imp output f input#1
connect f output d input
connect d output f input#2
connect f output p input
```

This schematic will work if we resolve all the ports to `FLOAT` (the output type of Impulse). But if we insist on both Fork inputs being resolved before we assign a type to the Fork output, we will be unable to resolve the schematic. So, once Pass 1 has recursively traversed the schematic and concluded that it can't yet assign a type to Fork's `input#2`, it uses the `FLOAT` type found at `input#1` to resolve the type of the Fork portholes. Further recursion then propagates this result around the schematic. This rather baroque-looking algorithm does have some simple properties. In particular, all members of an equivalence set are guaranteed to be assigned the same preferred type; an error will be reported if this is not possible. In some domains it is important that members of an equivalence set have the same resolved type, not just the same preferred type. (For example, the CGC Fork star fails if this is not so, because its various portholes are all just aliases for a single variable.) The domain can check this by seeing whether preferred type equals resolved type for all portholes. If the types are not the same, it can either report an error or splice in a type-conversion star to make them the same.

Note:

It might be better to cause this to happen on a per-star-type basis, not a per-domain basis, since one can imagine that some CG blocks would need strict type equality of portholes while others would not. This improvement is not currently implemented. The porthole type resolution algorithm is dependent on the notion that every porthole is connected to just one other porthole. If class Geodesic is ever redesigned to support multiple connections directly, some work would be needed. A likely tactic is to move some or all of the resolution work into Geodesic. A one-to-one Geodesic could enforce the same behavior described above, but one-to-many or many-to-many Geodesics would need different, possibly domain-dependent behavior.

6.11 Changes since Ptolemy0.6

Ptolemy 0.7 introduces several changes related to porthole type resolution. Older versions used a much simpler `ANYTYPE` resolution algorithm, which essentially amounted to just pass 2 ("feed back") of the present method. That had the serious deficiency that it couldn't decide what to do with fork stars feeding inputs of multiple types. For example, a fork star receiving `INT` and outputting to `INT` and `FLOAT` portholes would lead to a type resolution error. In essence, the old code insisted on being able to push type conversions back across a fork, and would fail if it couldn't assign the same type to all the fork outputs. The new algorithm solves this problem by delaying type conversions until after a fork. Occasionally this will introduce some inefficiency. For example, if a fork receives `INT` and feeds two `FLOAT`s, the new method leads to a type conversion being done separately on each fork output, whereas the old method would have generated only one conversion at the fork input. The improved ease of use of the new method is judged well worth this loss.

Formerly, the member ports of a multiporthole were always constrained to have the same resolved type. This is no longer true, since it gets in the way for polymorphic stars. But if a multiporthole is tied to another porthole via `inheritTypeFrom`, then each member porthole will still be constrained to match the type of that other porthole, at least in terms of preferred type. Formerly, the HOF stars acted during the galaxy "setup" phase, in which each block within the galaxy receives its setup call. This proved inadequate because porthole type resolution is done during setup; by the time a HOF star acted, the types of the portholes connected to it would already have been resolved. For example, a HOFNop star formerly constrained all the particles passing through it to be of the same type, because the porthole resolver insisted on being able to choose a unique type for the HOFNop's output porthole, even though that porthole is just a dummy that won't even exist at runtime. In Ptolemy 0.7, a "preinitialize" phase has been added so that HOF stars can rewire the galaxy and remove themselves before any block setup or porthole type resolution occurs. The constraints on porthole types are then only those resulting from the rewired schematic.