Chapter 7. Particles and Messages

Authors: *Joseph T. Buck*

7.1 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.

7.2 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. In general, we permit this only for Particles of the same type, and otherwise assert an error. But the arithmetic particle types invoke type conversion, via the virtual casting operators, so as to allow assignment from other arithmetic particle types. Without this exception, useful cases such as forking an INT output to INT and FLOAT inputs would fail in the simulation domains (because the fork stars use particle assignment).

```
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.

7.3 Arithmetic Particle classes

There are three standard arithmetic Particle classes: IntParticle, FloatParticle, and ComplexParticle. 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".

7.4 The Heterogeneous Message Interface

The heterogeneous message interface is a mechanism to permit messages of arbitrary type (ob-

jects 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.

7.4.1 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.

7.4.2 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 base class 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 overridden 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 base class. Exception: the base class 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 overridden 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.

7.4.3 Class MessageParticle

MessageParticle is a derived type of Particle 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.

7.5 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.`