Chapter 16. Base Code Generation Domain and Supporting Classes

Authors: *Soonhoi Ha*

Other Contributors: *Michael C. Williamson*

This chapter explains the base classes for code generation, which are found in the `$PTOLEMY/src/domains/cg/kernel` directory. Not all classes in that directory are covered in this document. We instead concentrate on how to generate and organize the code, and which methods to use. There is a basic code generation domain, designated CG, from which other code generation domains are derived. The CG domain can be used by itself for the purpose of studying issues in control constructs and scheduling, without needing to generate code in any particular programming language.

A segment of code is formed in an instance of class CodeStream. Each CGTarget will have a list of CodeStreams, and will assemble them to generate the final code. A CGStar uses instances of class CodeBlock to form a code segment, which can be added to a CodeStream of the CGTarget after some processing.

A set of macros are defined which a star programmer may use in order to refer to variables without being concerned about resource allocation. For example, we may refer to the portholes of a star without knowing what physical resources are allocated to them.

16.1 Class CodeStream

Class CodeStream is publicly derived from class StringList, and is used to make a sequential stream of code. In class CGTarget, a base target class for code generation, there are two CodeStreams: `myCode` and `procedures`.

```
CodeStream myCode;
CodeStream procedures;
```

These are protected members of class CGTarget. They are the default entries in `codeStringLists`, the list of code streams that CGTarget maintains.

```
CodeStreamList codeStringLists;
```

This is a protected member of class CGTarget. We can add a CodeStream to `codeStringLists` by using the following method of class CGTarget:

```
void addStream(const char* name, CodeStream* code);
```

This is a public method of class CGTarget. The first argument is the name of the CodeStream, and the second argument is a pointer to the CodeStream. This method should be called in the constructor of a target class. If a target attempts to add a CodeStream with an existing name, an error will be signaled.

`CodeStream* getStream(const char* name=NULL);`

This is a public method of class CGTarget. This method returns a pointer to the CodeStream with the given name. If no stream with the given name is found, this method returns `NULL`. If *name*`=NULL`, a pointer to `defaultStream` is returned. Class CGStar has a corresponding method to get the CodeStream with the given name from the CGStar's target.

The following method allows CGStars to construct a new CodeStream and add it to the CGTarget's list of CodeStreams. Some of the possible uses for this method are:

- a group of CGStars can build a procedure together
- a CGStar can add control flow constructs at the end of the `mainLoop` code

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

This is a public method of class CGTarget. There is a corresponding protected method of CGStar. This method adds a new CodeStream with the given name to the `codeStringLists` member of the CGTarget, and returns a pointer to the new CodeStream.

Now we will explain the public methods and members of class CodeStream.

`int put(const char* code, const char* name=NULL);`

This method puts the given segment of code at the end of the CodeStream. Optionally, the name of the code segment can be given. If *name*`=NULL,` we append the code unconditionally. Otherwise, we check to see if code with the same name has already ben added, by examining the `sharedNames` member of the CodeStream. If no code segment with the same name is found, the code segment is appended. This method returns `TRUE` if code was successfully added to the stream, `FALSE` otherwise.

`UniqueStringList sharedNames;`

This is a public member of class CodeStream. It is used to store the names of code segments added by name to the CodeStream. Class UniqueStringList is privately derived from class StringList.

`void initialize();`

This is a public method of class CodeStream. It is used to initialize both the code list and `sharedNames.`

`int isUnique(const char* name);`

This is a public method of class UniqueStringList. This method returns `FALSE` if the argument string already exists in the UniqueStringList. If not, then the method adds the string to the list and returns `TRUE`.

Class CodeStreamList contains a list of CodeStreams. It is publicly derived from class NamedList since each CodeStream is assigned a name. There are four public methods in the CodeStreamList class:

```
int append(CodeStream* stream, const char* name);
int add(const char* name, CodeStream* stream);
CodeStream* get(const char* name) const;
int remove(const char* name);
```

The first two methods append a CodeStream to the list. They differ from each other in the order of arguments. The third method returns a CodeStream with the given name while the last method removes the CodeStream with the given name from the list.

16.1.1 Class NamedList

Class NamedList is privately derived from class SequentialList, and is used to make a list of objects with names. It has a default constructor. The destructor deletes all objects in the list. There are no protected members in this class.

```
int append(Pointer object, const char* name);
void prepend(Pointer object, const char* name);
```

These methods put an object, *object*, with name *name* at the end and the beginning of the list, respectively. In the first method, we may not append multiple objects with the same name. If an object with the same name exists in the list, FALSE is returned. On the other hand, the second method allows multiple objects with the same name to be prepended. Only the most recently prepended object will be visible.

```
Pointer get(const char* name=NULL);
```

This method returns the object with the given name. If no name is given, it returns the object at the head of the list. If no object is found, it returns NULL.

```
int remove(const char* name=NULL);
```

This method removes the object with the given name. If no name is given, it removes the first object at the head of the list. If no object is found it returns FALSE, otherwise it returns TRUE.

There is an iterator class associated with the NamedList class, called NamedListIter. It returns a pointer to the next object in the list as it iterates through the list.

16.2 Class CodeBlock and Macros

Class CodeBlock stores a pointer to text in its constructor.

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

It is up to the programmer to make sure that the argument text lives as long as the code-block is used.

There are four public methods defined to access the text:

```
void setText(char* line);
const char* getText();
operator const char*();
void printCode();
```

The first method sets the text pointer in the CodeBlock. The next two methods return the text this CodeBlock points to. The last method prints the code to the standard output.

A star programmer uses the `codeblock` directive in the preprocessor language file to define a block of text. In a CodeBlock, the programmer uses the following macros in order to refer to the star ports and variables without needing to be concerned about resource manage-

ment or name conflicts:

`$val(`*`name`*`)`
Value of a state

`$size(`*`name`*`)`
Buffer size of a state or a porthole

`$ref(`*`name`*`)`
Reference to a state or a porthole

`$ref(`*`name, offset`*`)`
Reference with offset

`$label(`*`name`*`)`
Unique label inside a codeblock

`$codeblockSymbol(`*`name`*`)`
Another name for $label

`$starSymbol(`*`name`*`)`
Unique label inside a star

`$sharedSymbol(`*`list`*`, `*`name`*`)`
Unique label for set list, name pair

These macros are resolved into code after resources are allocated or unique symbols are generated.

A CodeBlock defined in a CGStar is put into a CodeStream of the target by the following methods of the CGStar class:

```
int addCode(const char* code, const char* stream=NULL,
            const char* name=NULL);
int addProcedure(const char* code, const char* name);
```

These are protected methods of class CGStar. The first method puts a segment of code, *`code,`* at the end of the target's CodeStream with name *`stream`*. If the name of the CodeStream is not given, the method uses the `myCode` stream of the target. The second method uses the `procedure` CodeStream of the target. The argument *`name`* of both methods is optionally used to specify the name of the code. If the code is successfully added, the methods return `TRUE`, otherwise they return `FALSE`. Before putting the code at the end of the CodeStream, the code is processed to resolve any macros by the application of the `processCode` method:

```
StringList processCode(CodeBlock& cb);
StringList processCode(const char* code);
```

These methods are both protected and essentially equivalent since the first method calls the second method. They scan the code, word by word, and copy it into a StringList. If a macro is found, the macro is expanded through a call to `expandMacro` before being copied to the StringList. Testing can be done to check whether a word is a macro or not by comparing the first character with the result of the following method:

```
virtual char substChar() const;
```
This method is a virtual protected method of class CGStar. It is used to return the special character that marks the beginning of a macro in a code block. In the CGStar class, it returns the dollar sign character, $.

```
virtual StringList expandMacro(const char* func,
                               const StringList& argList);
```
This is a virtual protected method of class CGStar. It is used to expand a macro with the given name *func.* The argument list must be passed by reference so that the StringList will not be consolidated. It is virtual so that derived classes can define more macros. A macro is identified by the following method:

```
int matchMacro(const char* func, const StringList& argList,
               const char* name, int argc);
```
This protected method of class CGStar returns TRUE if the first argument *func* matches with the third argument *name*, and the number of arguments in *argList* is the same as the count *argc*.

Based on the particular macro being applied, one of the following protected methods may be used to expand the macro:

```
virtual StringList expandVal(const char* name);
StringList expandSize(const char* name);
virtual StringList expandRef(const char* name);
virtual StringList expandRef(const char* name, const char* offset);
```
The first three methods expand the $val, $size, and $ref macros. The fourth method expands the $ref macro when it has two arguments. These virtual methods should be redefined in derived classes. In particular, the last two methods must be redefined in derived classes because in class CGStar they generate error messages. The other macros deal with unique symbols within the scope of a code block, within a star, and within a set of symbols. More will be said about these in the next subsection .

When an error is encountered while expanding macros or processing code blocks, the following methods should be called to generate an error message:

```
void macroError(const char* func, const StringList& argList);
void codeblockError(const char* p1,  const char* p2="");
```
The arguments of the second method provide the text of the error message.

16.3 Class SymbolList and Unique Symbol Generation

In order to generate a unique symbol within a scope, a list of symbols should be made for that scope. For example, the CGStar class has two protected members which are SymbolLists, `starSymbol` and `codeblockSymbol`.

```
SymbolList starSymbol;
SymbolList codeblockSymbol;
```
Class SymbolList is derived from class BaseSymbolList. Class BaseSymbolList is privately derived from class NamedList. A BaseSymbolList keeps two private members which are used to create a unique name for each symbol in the list: a separator and a counter:

```
BaseSymbolList(char sep='_', int* count=NULL);
```
The first argument of the constructor is used to set the separator, and the second argument is used to set the pointer of the count variable. These two variables can be set independently by invoking the following methods:

```
void setSeparator(char sep);
void setCounter(int* count);
```
When we append or insert a new symbol into the list, we create a unique name for that symbol by appending a separator followed by the counter value to the argument symbol, and then return the unique name:

```
const char* append(const char* name);
const char* prepend(const char& name);
const char* get(const char* name=NULL);
```
This last method returns the unique symbol with the given name. If no name is given, it returns the first symbol in the list.

```
int remove(const char* name=NULL);
```
This method removes the unique symbol with the given name. If no name is given, it removes the first symbol in the list. It returns `FALSE` if no symbol is removed.

Symbols in the list are deleted in the destructor, and in the following method:
```
void initialize();
```

The public method
```
Stringlist symbol(const char* string)
```
makes a unique symbol from the supplied argument by adding the separator and a unique counter value to the argument string.

Class SymbolList is privately derived from class BaseSymbolList with the same constructor and a default destructor. Class SymbolList uncovers only three methods of the base class:
```
BaseSymbolList::setSeparator;
BaseSymbolList::setCounter;
BaseSymbolList::initialize;
```

Class SymbolList adds one additional method:
```
const char* lookup(const char* name);
```
If a unique symbol with the given name exists, this method returns that unique symbol. Otherwise, it creates a unique symbol with that name and puts it into the list.

Recall that the CGStar class has two SymbolLists. The macros `$codeblockSymbol`, `$label`, and `$starSymbol` are resolved by the `lookUp` method of the `codeblockSymbol` and `starSymbol` SymbolLists, based on the scope of the symbol. If the symbol already exists in the SymbolList, it returns that unique symbol. Otherwise, it creates a unique symbol in the scope of interest.

If we want to generate a unique symbol within the file scope, we use a scoped symbol list defined in the target class.

```
ScopedSymbolList sharedSymbol;
```

It is a protected member of the CGTarget class. Class ScopedSymbolList is privately derived from class NamedList to store a list of SymbolLists. It has the same constructor as the base class.

```
void setSeparator(char set);
void setCounter(int* count);
```

These methods in class ScopedSymbolList are used to set the separator and the counter pointer of all SymbolLists in the list.

```
const char* lookup(const char* scope, const char* name);
```

In this method, the first argument determines the SymbolList in the list named *scope,* and the second argument determines the unique symbol within that SymbolList. If no SymbolList is found with the given name, we create a new SymbolList and insert it into the list.

The SymbolLists in the list are deleted in the destructor and in the following method:

```
void initialize();
```

Now we can explain how to expand the last macro defined in the CGStar class: $sharedSymbol. The first argument of the macro determines the StringList and the second argument accesses the unique string in that StringList. It is done by calling the following protected method in the CGStar class:

```
const char* lookupSharedSymbol(const char* scope, const char* name);
```

This method calls the corresponding method defined the CGTarget class.

The CGTarget class has another symbol list:

```
SymbolStack targetNestedSymbol;
```

It is a protected member used in generating unique nested symbols. Class SymbolStack is privately derived from class BaseSymbolList. It has the same constructor as the base class and has a default destructor.

For stack operation, class SymbolStack defines the following two methods:

```
const char* push(const char* tag="L");
StringList pop();
```

These methods push the symbol with given name onto the top of the list and pop the symbol at the top of the list off of the list, respectively.

This class also exposes several methods of the base class:

```
BaseSymbolList::get;
BaseSymbolList::setSeparator;
BaseSymbolList::setCounter;
BaseSymbolList::initialize;
BaseSymbolList::symbol;
```

In this section, we have explained various symbol lists. The separator and the counter are usually defined in the CGTarget class:

```
char separator;
int counter;
```

The first is a public member in class CGTarget, and is set in the constructor. The second is a private member in class CGTarget, and is initialized to zero in the constructor. The counter value is accessed through the following public method:

```
int* symbolCounter();
```

16.4 Class CGGeodesic and Resource Management

When we generate assembly code, we have to allocate memory locations to implement the portholes and states of each star. For high-level language generation, we assign unique identifiers to them. It is rather easy to allocate resources for states since state requirements are visible from the star definition: type, size and name. In this section, we will focus on how to determine the buffer size for the porthole connections.

We allocate a buffer for each connection. We do not assume in the base class, however, that the buffer is owned by the source or by the destination porthole. Instead, we use methods of the CGGeodesic class. Before determining the buffer sizes, we obtain information about how many samples are accumulated on each CGGeodesic by simulating the schedule. This is for the case of synchronous dataflow (SDF) semantics with static scheduling.

The minimum buffer requirement of a connection may be determined by considering only local information about the connection:

```
int minNeeded() const;
```

This method returns that minimum buffer size. It is a protected member of class CGGeodesic.

We do not want to allocate buffers for connections when it is unnecessary. For example, the output portholes of a Fork star can share the same resource with the Fork star's input porthole. A Gain star with unity gain is another trivial example. Therefore, we pay special attention to stars of type Fork. Without confusion, we refer to a star as a *Fork* star if its outputs can share the same resource with its input. In the CGStar class, we provide the following methods:

```
int isItFork();
void isaFork();
virtual void forkInit(CGPortHole& input, MultiCGPortHole& output);
virtual void forkInit(CGPortHole& input, CGPortHole& output);
```

The first is a public method of class CGStar. The rest are protected methods of class CGStar. The first method queries whether the star is a Fork star. The second method is used to declare that the star is a Fork star. If it is, we can call either one of the last two methods, based on whether the output is a MultiPortHole or not. In those methods, we shift delays from a Fork's input port to the output ports, and set the `forkSrc` pointer of the output ports to point to the Fork's input port. The Fork's input port keeps a list of the output ports in its `forkDests` member. We apply this procedure recursively in the case of cascaded Forks.

```
CGPortHole* forkSrc;
SequentialList forkDests;
```
These are protected members of class CGPortHole. The first one is set by the following public method:

```
void setForkSource(CGPortHole* p, int cgPortHoleFlag=TRUE);
```
The first argument is the input porthole of the Fork star and the port this is being called on should be an output porthole when we call this method.

```
int fork() const;
```
This is a public method of class CGPortHole which returns `TRUE` if it is an input porthole of a Fork star.

Class CGGeodesic provides two methods to return the Fork input port if it is at a Fork output port. Otherwise these methods return `NULL`.

```
CGPortHole* src();
const CGPortHole* src() const;
```
These two methods are protected and differ from each other in their return type.

Now we will explain more of the methods of class CGGeodesic.

```
int forkType() const;
```
This public method of class CGGeodesic indicates the type of the current CGGeodesic. If it is at a Fork input, it is `F_SRC.` If it is at a Fork output, it is `F_DEST.`

```
int forkDelay() const;
```
This public method of class CGGeodesic returns the amount of delay from the current Geodesic up to the fork buffer that this Geodesic refers to. If it is not associated with a fork buffer, it returns 0.

We do not allocate a buffer to a CGGeodesic if it is `F_DEST.`

```
int localBufSize() const;
int bufSize() const;
```
The above public methods of class CGGeodesic return the buffer size associated with this CGGeodesic. While the first method returns 0 if the CGGeodesic is at a Fork output, the second method returns the size of the fork buffer. The actual computation of the buffer size is done by applying the following method:

```
virtual int internalBufSize() const;
```
This protected method of class CGGeodesic returns 0 with an error message if the schedule has not yet been run. If this CGGeodesic is a `F_SRC,` the minimum size is set to the maximum buffer requirements over all fork destinations. If there are delays or if old values are used, we may want to use a larger size so that compile-time indexing is supportable. The buffer size must divide the total number of tokens produced in one execution. To avoid modulo addressing, we prefer to use the *LCM* value of the number of samples consumed and produced during one iteration of the schedule. Since this may be wasteful, we check the extra buffer size required for linear addressing with the `wasteFactor.` If the waste ratio is larger then `wasteFactor,` we give up on linear addressing.

```
virtual double wasteFactor() const;
```
In the CGGeodesic class, this method returns 2.0. If a derived class wants to enforce linear addressing as much as possible, it should set the return value to be large. To force the minimum buffer memory size to be used, the return value should be set to 1.0.

```
void initialize();
```
This public method of class CGGeodesic initializes the CGGeodesic.

Refer to class CGPortHole for more information on resource management.

16.5 Utility Functions

There are several utility functions defined in the CG domain for aiding in code generation. Here we describe just a few of them:

```
char* makeLower(const char* name);
int rshSystem(const char* hostname, const char* command,
              const char* directory=NULL);
```
The above functions are defined in the file CGUtilities.h. The first method returns a dynamically allocated string that is a lower-case version of the argument string. The second method is used to execute a remote shell command, *command,* in the *directory* on the machine *hostname.* We use the `xon` command instead of `rsh` in order to preserve any X-Window environment variables.

16.6 Class CGStar

In this section, we will explain additional class CGStar members and methods not described above in this chapter. Class CGStar has a constructor with no arguments. Class CGStar is derived from class DynDFStar, and not from class SDFStar, so that BDF and DDF code generation may be supported in the future.

There is an iterator to enumerate the CGPortHoles of a CGStar: class CGStarPortIter. The `next()` and `operator++` methods return type `CGPortHole*`.

16.6.1 CGStar Protected Methods and Members

Protected members related to CodeStream, SymbolList, and resource management can be found in earlier sections of this chapter.

```
virtual void outputComment(const char* msg, const char* stream=NULL);
```
This method adds a comment *msg* to the target *stream.* If no target stream is specified, the `myCode` stream is used.

```
StringList expandPortName(const char* name);
```
If the argument specifies the name of a MultiPortHole, the index may be indicated by a State. In this case, this method gets the value of the State as the index to the MultiPortHole and returns a valid MultiPortHole name. This method is used in the `expandSize` method.

```
void advance();
```
This method updates the offset variable of all PortHoles of the CGStar by the number of

samples consumed or produced. It calls the `advance` method of each PortHole.

`IntState procId;`
This is an integer state to indicate processor assignment for parallel code generation. By default, the value is -1 to indicate that the star is not yet assigned.

`int dataParallel`
This is a flag to be set if this star is a wormhole or a parallel star.

`Profile* profile;`
This is a pointer to a Profile object, which can be used to indicate the local schedule of a data parallel star or macro actor. If it is not a parallel star, this pointer is set `NULL`.

`int deferrable();`
When constructing a schedule for a single processor, we can defer the firing of a star as long as possible in order to reduce the buffer requirements on every output arc. In this method, we never defer a Fork star, and always defer any non-Fork star that feeds into a Fork. This prevents the resulting fork buffer from being larger than necessary, because new tokens are not added until they must be.

16.6.2 CGStar Public Methods

`const char* domain() const;`
`int isA(const char* class);`
The first method returns `"CG"`. The second method returns `TRUE` if the argument *`class`* is `CGStar` or a base class of CGStar.

`int isSDF() const;`
Returns `TRUE` if it is a star with SDF semantics (default). For BDF and DDF stars, it will return `FALSE`.

`virtual void initCode();`
This method allows a star to generate code outside the main loop. This method will be called after the schedule is created and before the schedule is executed. In contrast, the `go()` method is called during the execution of the schedule, to form code blocks into a main loop body.

`int run();`
In CG domains, this method does not perform any actual data movement, but executes the `go()` method followed by the `advance()` method.

`CGTarget* cgTarget();`
`int setTarget(Target* t);`
These methods get and set the pointer to the target to which this star is assigned. When we set the target pointer, we also initialize the SymbolLists and the CodeStream pointers. If this method is successful, it returns `TRUE`, otherwise it returns `FALSE`.

`virtual int isParallel() const;`
`virtual Profile* getProfile(int ix=0);`

The first method returns `TRUE` if this star is a wormhole or a parallel star. If it is parallel, the second method returns the pointer to a Profile, indexed by the argument. A parallel star stores its internal scheduling results in a Profile object .

```
int maxComm();
```
Returns the maximum communication overhead with all ancestors. It calls the `commTime` method of the target class to obtain the communication cost.

```
virtual void setProcId(int i);
virtual int getProcId();
```
These methods set and get the processor ID to which this star is assigned.

16.7 Class CGPortHole

Class CGPortHole is derived from class DynDFPortHole in order to support non-SDF dataflow stars as well as SDF stars. Methods related to Fork stars are described in a the section above on Resource Management .

In this section, we will categorize the members and methods of CGPortHole into four categories: buffer management, buffer embedding, geodesic switching, and others.

16.7.1 Buffer Management

A CGPortHole is connected to a buffer after resource allocation. A CGPortHole maintains an offset index into the buffer in order to identify the current position in the buffer where the porthole will put or get the next sample:

```
int offset;
```
This is a protected member used for indexing into the buffer connected to this port.

The methods described in this subsection are all public:

```
unsigned bufPos() const;
```
Returns `offset`, the offset position in the buffer.

```
virtual int bufSize() const;
virtual int localBufSize() const;
```
Both methods returns the size of buffer connected to this porthole. In the CGPortHole base class, they call the corresponding methods of class CGGeodesic. Recall that the second method returns 0 when it is a Fork output. If a porthole is at the wormhole boundary, both return the product of the sample rate and the repetition count of the parent star.

```
virtual void advance();
```
This method is called by `CGStar::advance()`. After the parent star is executed, we advance the offset by the number of samples produced or consumed. The offset is calculated modulo the buffer size, so that it is wrapped around if it reaches the boundary of the buffer.

16.7.2 Buffer Embedding

As a motivating example, let's consider a DownSample star. If we allocate separate buffers to

the input and output ports, the buffer size of the input port will be larger than that of the output port. Also, we will need to perform an unnecessary copy of samples. We can improve this situation by allocating one buffer at the input site and by indicating that a subset of that buffer is the image of the output buffer. We call this relationship *embedding*: the larger input buffer is *embedding* the smaller output buffer, and the smaller output buffer is *embedded* in the larger input buffer. Unlike the Fork buffer, the sizes of input and output embedded buffers are different from each other. Therefore, we must specify at which position the embedded buffer begins in the larger embedding buffer. We use this embedding relationship to implement Spread and Collect stars in the CGC domain, without increasing the buffer requirements. For example, the output buffers of a Spread star are embedded in the input buffer of the star, starting from different offsets.

```
CGPortHole* embeddedPort;
int embeddingFlag;
```

These are protected members to specify embedding relationships. The first one points to the embedding port which this PortHole is embedded in. The second member indicates the starting offset of embedding. The last member indicates whether this porthole is an embedding port or not.

The following are public methods related to embedding.

```
CGPortHole* embedded();
int whereEmbedded();
int embedding();
```

These methods return the protected members described above, respectively.

```
void embed(CGPortHole& p, int i=-1);
```

This method establishes an embedding relationship between this port and the argument port *p*. This porthole becomes an embedding porthole, and the argument porthole becomes an embedded porthole. The second argument specifies the starting offset.

```
void embedHere(int offset);
```

This method, when called on an embedded porthole, changes the starting offset its embedded buffer in the embedding buffer.

16.7.3 Geodesic Switching

In the specification block diagram, a PortHole is connected to a Geodesic. In code generation domains, we usually allocate one resource to the Geodesic so that the Geodesic's source and destination ports can share the same resource (Note that this is not a strict requirement). After resource allocation, we may want to alias a porthole to another porthole, and therefore associate it with a resource other than the allocated resource. To do that, we switch the pointer of the Geodesic to another Geodesic.

```
virtual void switchGeo(Geodesic* g);
virtual void revertGeo(Geodesic* g);
```

Both methods set the Geodesic pointer to the argument *g*. There is a flag to indicate whether this port has switched its Geodesic or not. The first method sets the flag to TRUE while the second method resets the flag to FALSE. Both methods are virtual since in derived classes we may need to redefine the behavior, perhaps by saving the original Geo-

desic, which is not the default behavior. The flag is queried by:

```
int switched() const;
```

If the Geodesic is switched in this port, we have to reset the geodesic pointer of this port to `NULL` in the destructor in order to prevent attempts to delete the same Geodesic multiple times. Also, we have to make sure that both ends of a Geodesic do not switch their Geodesic, in order to prevent orphaning the geodesic and causing a memory leak.

16.7.4 Other CGPortHole Members

Class CGPortHole has a constructor with no argument which resets the member variables. In the destructor, we clear the `forkDests` list and remove the pointer to this porthole from the `forkDests` list of the `forkSrc` port. All members described in the subsection are public.

```
CGGeodesic& cgGeo() const;
```

This method returns a reference to the Geodesic after type casting.

```
void forceSendData();
void forceGrabData();
```

These methods put and get samples to and from the Geodesic at the wormhole boundary. They are used when the inside code generation domain communicates by the wormhole mechanism.

16.7.5 CGPortHole Derived Classes

Class InCGPort and class OutCGPort are publicly derived from class CGPortHole. They are used to indicate by class type whether a porthole is an input port or an output port.

Class MultiCGPort is derived from class MultiDFPort. It has a protected member `forkSrc` to point to the Fork input if its parent star is a Fork star. It has a default destructor.

```
CGPortHole* forkSrc;
```

There are two public methods related to this protected member:

```
void setForkBuf(CGPortHole& p);
void forkProcessing(CGPortHole& p);
```

The first method sets `forkSrc` to point to the argument port. The second method sets the `forkSrc` pointer of the argument port to point to the `forkSrc` of this MultiCGPort.

Two classes are publicly derived from MultiCGPort: MultiInCGPort and MultiOutCGPort. They both have the following public method:

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

This method creates an InCGPort or an OutCGPort depending on whether it is an input or an output MultiCGPort.