

Ross Bencina

This chapter explores the implementation internals of scsynth, the server process of SuperCollider 3, which is written in C++. This chapter is intended to be useful to people who are interested in modifying or maintaining the scsynth source code and also to those who are interested in learning about the structure and implementation details of one of the great milestones in computer music software. By the time you've finished this chapter, you should have improved your understanding of how scsynth does what it does and also have gained some insight into why it is written the way it is. In this chapter sometimes we'll simply refer to scsynth as "the server." "The client" usually refers to slang or any other program sending OSC commands to the server. Although the text focuses on the server's real-time operating mode, the information presented here is equally relevant to understanding scsynth's non-real-time mode. As always, the source code is the definitive reference and provides many interesting details which space limitations didn't allow to be included here.

Wherever possible, the data, structure, and function names used in this chapter match those in the scsynth source code. However, at the time of writing there was some inconsistency in class and structure naming. Sometimes you may find that the source file, the class name, or both may have an `SC_` prefix. I have omitted such prefixes from class and function names for consistency.

Also note that I have chosen to emphasize an object-oriented interpretation of scsynth using UML diagrams to illuminate the code structure, as I believe scsynth is fundamentally object-oriented, if not in an idiomatically C++ way. In many cases structs from the source code appear as classes in the diagrams. Where appropriate, I have taken the liberty to interpret inheritance where a base struct is included as the first member of a derived struct. However, I have resisted the urge to translate any other constructs (such as the pseudo member functions mentioned below). All other references to names appear here as they do in the source code.

Now that formalities are completed, in the next section we set out on our journey through the scsynth implementation with a discussion of scsynth's coding style. Following that, we consider the structure of the code which implements what I call the

scsynth domain model: Nodes, Groups, Graphs, GraphDefs, and their supporting infrastructure. We then go on to consider how the domain model implementation communicates with the outside world; we consider threading, interthread communications using queues, and how scsynth fulfills real-time performance constraints while executing all of the dynamic behavior offered by the domain model. The final section briefly highlights some of the fine-grained details which make scsynth one of the most efficient software synthesizers on the planet. scsynth is a fantastically elegant and interesting piece of software; I hope you get as much out of reading this chapter as I did in writing it!

26.1 Some Notes on scsynth Coding Style

scsynth is coded in C++, but for the most part uses a “C++ as a better C” coding style. Most data structures are declared as plain old C structs, especially those which are accessible to unit plug-ins. Functions which in idiomatic C++ might be considered member functions are typically global functions in scsynth. These are declared with names of the form `StructType_MemberFunctionName(StructType *s[, ...])`, where the first parameter is a pointer to the struct being operated on (the “this” pointer in a C++ class). Memory allocation is performed with custom allocators or with `malloc()`, `free()`, and friends. Function pointers are often used instead of virtual functions. A number of cases of what can be considered inheritance are implemented by placing an instance of the base class (or struct) as the first member of the derived struct. There is very little explicit encapsulation of data using getter/setter methods.

There are a number of pragmatic reasons to adopt this style of coding. Probably the most significant is the lack of an Application Binary Interface (ABI) for C++, which makes dynamically linking with plug-ins using C++ interfaces compiler-version-specific. The avoidance of C++ constructs also has the benefit of making all code operations visible, in turn making it easier to understand and predict the performance and real-time behavior of the code.

The separation of data from operations and the explicit representation of operations as data-using function pointers promotes a style of programming in which types are composed by parameterizing structs by function pointers and auxilliary data. The use of structs instead of C++ classes makes it less complicated to place objects into raw memory. Reusing a small number of data structures for many purposes eases the burden on memory allocation by ensuring that dynamic objects belong to only a small number of size classes. Finally, being able to switch function pointers at runtime is a very powerful idiom which enables numerous optimizations, as will be seen later.

26.2 The scsynth Domain Model

At the heart of scsynth is a powerful yet simple domain model which manages dynamic allocation and evaluation of unit generator graphs in real time. Graphs can be grouped into arbitrary trees whose execution and evaluation order can be dynamically modified (McCartney, 2000). In this section we explain the main behaviors and relationships between entities in the domain model. The model is presented without concern for how client communication is managed or how the system is executed within real-time constraints. These concerns are addressed in later sections.

Figure 26.1 shows an implementation-level view of the significant domain entities in scsynth. Each class shown on the diagram is a C++ class or struct in the scsynth source code. SC users will recognize the concepts modeled by many of these classes. Interested readers are advised to consult the “ServerArchitecture” section of the Help files for further information about the roles of these classes and the exact operations which can be performed by them.

`World` is the top-level class which (with the exception of a few global objects) aggregates and manages the run-time data in the server. It is created by `World_New()` when scsynth starts up. An instance of `WorldOptions` is passed to `World_New()`. It stores the configuration parameters, which are usually passed to scsynth on the command line.

scsynth’s main task is to synthesize and process sound. It does this by evaluating a tree of dynamically allocated `Node` instances (near middle-left of figure 26.1), each of which provides its own `NodeCalcFunc` function pointer, which is called by the server to evaluate the `Node` at the current time step. `Node::mID` is an integer used by clients to identify specific `Nodes` in server commands (such as suspending or terminating the `Node`, or changing its location in the tree).

There are 2 subtypes of `Node`: `Graph` and `Group`. `Graph` is so named because it executes an optimized graph of `UGens`. It can be likened to a voice in a synthesizer or an “instrument” in a *Music N*-type audio synthesis language such as `Csound`. The `Graph` type implements the SuperCollider concept of a *Synth*. `Group` is simply a container for a linked list of `Node` instances, and since `Group` is itself a type of `Node`, arbitrary trees may be constructed containing any combination of `Group` and `Graph` instances; readers may recognize this as the *Composite* design pattern (Gamma et al., 1995). The standard `NodeCalcFunc` for a `Group` (`Group_Calc()` in `SC_Group.cpp`) simply iterates through the `Group`’s contained `Nodes`, calling each `Node`’s `NodeCalcFunc` in turn. Although most code deals with `Nodes` polymorphically, the `Node::mIsGroup` field supports discriminating between `Nodes` of type `Graph` and of `Group` at runtime. Any node can be temporarily disabled using the `/n_run` server command, which switches `NodeCalcFuncs`. When a `Node` is switched off, a `NodeCalcFunc` which does

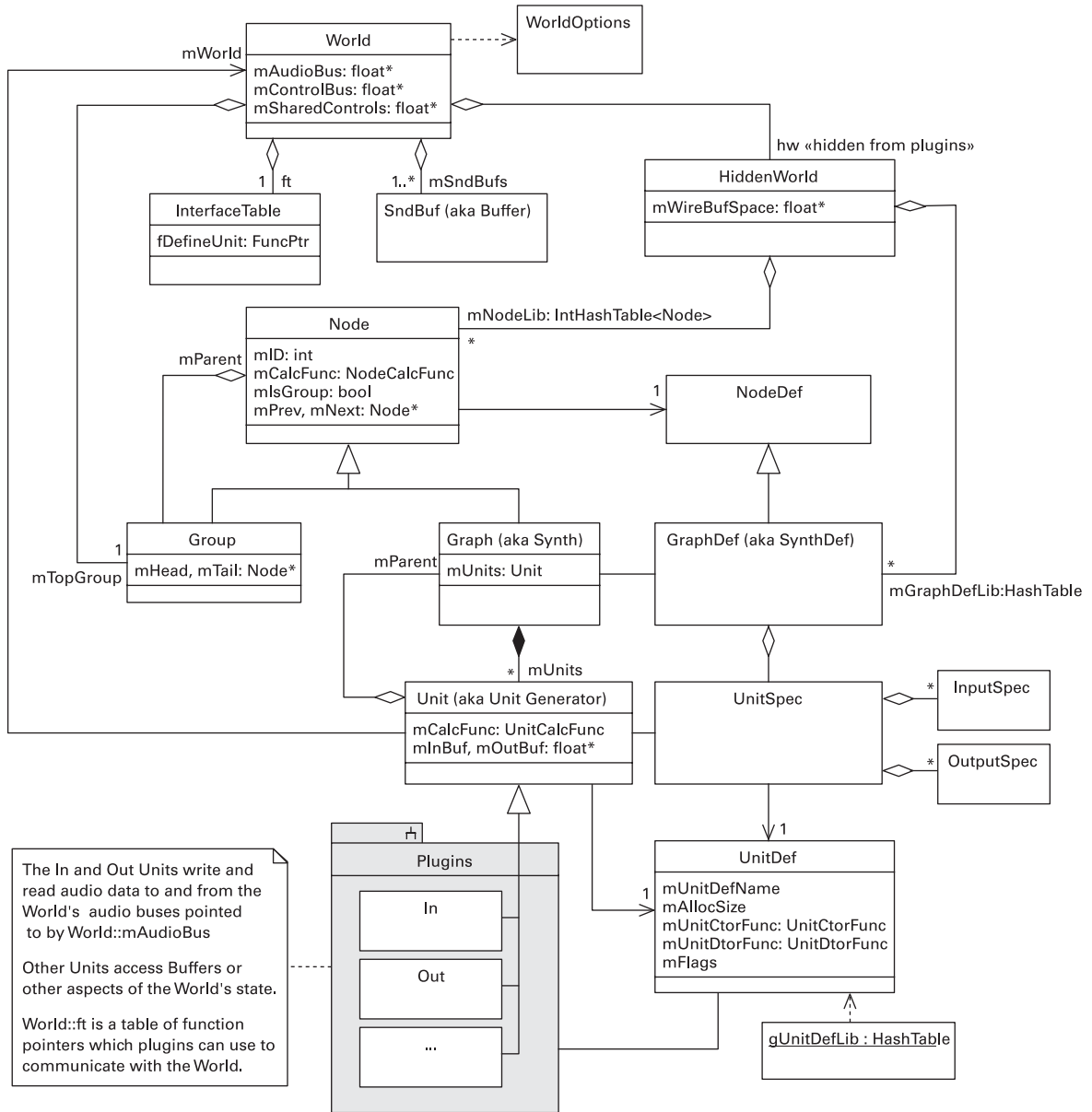


Figure 26.1
Class diagram of significant domain entities.

nothing is substituted for the usual one. Disabling a Group disables the whole tree under that Group.

A Graph is an aggregate of interconnected Unit subclasses (also known as Unit Generators or UGens). Unit instances are responsible for performing primitive audio DSP operations such as mixing, filtering, and oscillator signal generation. Each Graph instance is carved out of a single memory block to minimize the number of expensive calls to the memory allocator. Units are efficiently allocated from the Graph's memory block and evaluated by iterating through a linear array containing pointers to all of the Graph's Units. Each Unit instance provides a UnitCalcFunc function pointer to compute samples, which affords the same kind of flexibility as NodeCalcFunc described above. For example, many Units implement a form of self-modifying code by switching their UnitCalcFuncs on the fly to execute different code paths, depending on their state.

Graphs are instantiated using a GraphDef (Graph Definition), which defines the structure of a class of Graphs. The GraphDef type implements the SuperCollider concept of a *SynthDef*. A GraphDef includes both data for passive representation (used on disk and as communicated from clients such as slang), and optimized in-memory information used to efficiently instantiate and evaluate Graphs. GraphDef instances store data such as memory allocation size for Graph instances, Unit initialization parameters, and information about the connections between Units. When a new GraphDef is loaded into the server, most of the work is done in GraphDef_Read(), which converts the stored representation to the run-time representation. Aside from allocating and initializing memory and wiring in pointers, one of the main tasks GraphDef_Read() performs is to determine which inter-Unit memory buffers will be used to pass data between Units during Graph evaluation.

The stored GraphDef representation specifies an interconnected graph of named Unit instances with generalized information about input and output routing. This information is loaded into an in-memory array of UnitSpec instances where each Unit name is resolved to a pointer to a UnitDef (see below), and the Unit interconnection graph is represented by instances of InputSpec and OutputSpec. This interconnection graph is traversed by a graph-coloring algorithm to compute an allocation of inter-Unit memory buffers, ensuring that the minimum number of these buffers is used when evaluating the Graph. Note that the order of Unit evaluation defined by a GraphDef is not modified by scsynth.

scsynth's tree of Nodes is rooted at a Group referenced by World::mTopGroup. World is responsible for managing the instantiation, manipulation, and evaluation of the tree of Nodes. World also manages much of the server's global state, including the buses used to hold control and audio input and output signals (e.g., World::mAudioBus) and a table of SndBuf instances (aka *Buffers*) used, for example, to hold sound data loaded from disk. An instance of World is accessible to Unit plug-ins via Unit::mWorld

and provides `World::ft`, an instance of `InterfaceTable`, which is a table of function pointers which Units can invoke to perform operations on the World. An example of Units using World state is the `In` and `Out` units which directly access `World::mAudioBus` to move audio data between Graphs and the global audio buses.

`Unit` subclasses provide all of the signal-processing functionality of `scsynth`. They are defined in dynamically loaded executable “plug-ins.” When the server starts, it scans the nominated plug-in directories and loads each plug-in, calling its `load()` function; this registers all available Units in the plug-in with the World via the `InterfaceTable::fDefineUnit` function pointer. Each call to `fDefineUnit()` results in a new `UnitDef` being created and registered with the global `gUnitDefLib` hash table, although this process is usually simplified by calling the macros defined in `SC_InterfaceTable.h`, such as `DefineSimpleUnit()` and `DefineDtorUnit()`.

Some server data (more of which we will see later) is kept away from `Unit` plug-ins in an instance of `HiddenWorld`. Of significance here are `HiddenWorld::mNodeLib`, a hash table providing fast lookup of `Nodes` by integer ID; `HiddenWorld::mGraphDefLib`, a hash table of all loaded `GraphDefs`, which is used when a request to instantiate a new `Graph` is received; and `HiddenWorld::mWireBufSpace`, which contains the memory used to pass data between Units during `Graph` evaluation.

26.3 Real-Time Implementation Structure

We now turn our attention to the context in which the server is executed. This includes considerations of threading, memory allocation, and interthread communications. `scsynth` is a real-time system, and the implementation is significantly influenced by real-time requirements. We begin by considering what “real-time requirements” means in the context of `scsynth` and then explore how these requirements are met.

26.3.1 Real-Time Requirements

`scsynth`’s primary responsibility is to compute blocks of audio data in a timely manner in response to requests from the OS audio service. In general, the time taken to compute a block of audio must be less than the time it takes to play it. These blocks are relatively small (on the order of 2 milliseconds for current generation systems), and hence tolerances can be quite tight. Any delay in providing audio data to the OS will almost certainly result in an audible glitch.

Of course, computing complex synthesized audio does not come for free and necessarily takes time. Nonetheless, it is important that the time taken to compute each block is bounded and as close to constant as possible, so that exceeding timing constraints occurs only due to the complexity or quantity of concurrently active Graphs, not to the execution of real-time unsafe operations. Such unsafe operations include

- Algorithms with high or unpredictable computational complexity (for example, amortized time algorithms with poor worst-case performance)
- Algorithms which intermittently perform large computations (for example, pre-computing a lookup table or zeroing a large memory block at Unit startup)
- Operations which block or otherwise cause a thread context switch.

The third category includes not only explicit blocking operations, such as attempting to lock a mutex or wait on a file handle, but also operations which may block due to unknown implementation strategies, such as calling a system-level memory allocator or writing to a network socket. In general, any system call should be considered real-time unsafe, since there is no way to know whether it will acquire a lock or otherwise block the process.

Put simply, no real-time unsafe operation may be performed in the execution context which computes audio data in real time (usually a thread managed by the OS audio service). Considering the above constraints alongside the dynamic behavior implied by the domain model described in the previous section and the fact that scsynth can read and write sound files on disk, allocate large blocks of memory, and communicate with clients via network sockets, you may wonder how scsynth can work at all in real time. Read on, and all will be revealed.

26.3.2 Real-Time Messaging and Threading Implementation

SuperCollider carefully avoids performing operations which may violate real-time constraints by using a combination of the following techniques:

- Communication to and from the real-time context is mediated by lock-free First In First Out (FIFO) queues containing executable messages
- Use of a fixed-pool memory allocator which is accessed only from the real-time context
- Non-real-time safe operations (when they must be performed at all) are deferred and executed asynchronously in a separate “non-real-time” thread
- Algorithms which could introduce unpredictable or transient high computational load are generally avoided
- Use of user-configurable nonresizable data structures. Exhaustion of such data structures typically results in scsynth operations failing.

The first point is possibly the most important to grasp, since it defines the pervasive mechanism for synchronization and communication between non-real-time threads and the real-time context which computes audio samples. When a non-real-time thread needs to perform an operation in the real-time context, it enqueues a message which is later performed in the real-time context. Conversely, if code in the

real-time context needs to execute a real-time unsafe operation, it sends the message to a non-real-time thread for execution. We will revisit this topic on a number of occasions throughout the remainder of the chapter.

Figure 26.2 shows another view of the `scsynth` implementation, this time focusing on the classes which support the real-time operation of the server. For clarity, only a few key classes from the domain model have been retained (shaded gray). Note that `AudioDriver` is a base class: in the implementation different subclasses of `AudioDriver` are used depending on the target OS (`CoreAudio` for Mac OS X, `PortAudio` for Windows, etc.).

Figure 26.3 illustrates the run-time thread structure and the dynamic communication pathways between threads via lock-free FIFO message queues. The diagram can be interpreted as follows: thick rectangles indicate execution contexts, which are either threads or callbacks from the operating system. Cylinders indicate FIFO message queue objects. The padlock indicates a lock (mutex), and the black circle indicates a condition variable. Full arrows indicate synchronous function calls (invocation of queue-member functions), and half arrows indicate the flow of asynchronous messages across queues.

The FIFO message queue mechanism will be discussed in more detail later in the chapter, but for now, note that the `Write()` method enqueues a message, `Perform()` executes message-specific behavior for each pending message, and `Free()` cleans up after messages which have been performed. The `Write()`, `Perform()`, and `Free()` FIFO operations can be safely invoked by separate reader and writer threads without the use of locks.

Referring to figures 26.2 and 26.3, the dynamic behavior of the server can be summarized as follows:

1. One or more threads listen to network sockets to receive incoming OSC messages which contain commands for the server to process. These listening threads dynamically allocate `OSC_Packets` and post them to “The Engine,” using the `ProcessOSCPacket()` function, which results in `Perform_ToEngine_Msg()` (a `FifoMsgFunc`) being posted to the `mOscPacketsToEngine` queue. `OSC_Packet` instances are later freed, using `FreeOSCPacket()` (a `FifoFreeFunc`) by way of `MsgFifo::Free()`, via a mechanism which is described in more detail later.
2. “The Synthesis Engine,” or “Engine” for short (also sometimes referred to here as “the real-time context”), is usually a callback function implemented by a concrete `AudioDriver` which is periodically called by the OS audio service to process and generate audio. The main steps relevant here are that the Engine calls `Perform()` on the `mOscPacketsToEngine` and `mToEngine` queues, which execute the `mPerformFunc` of any messages enqueued from other threads. Messages in `mOscPacketsToEngine` carry `OSC_Packet` instances which are interpreted to manipulate the Node tree, instantiate

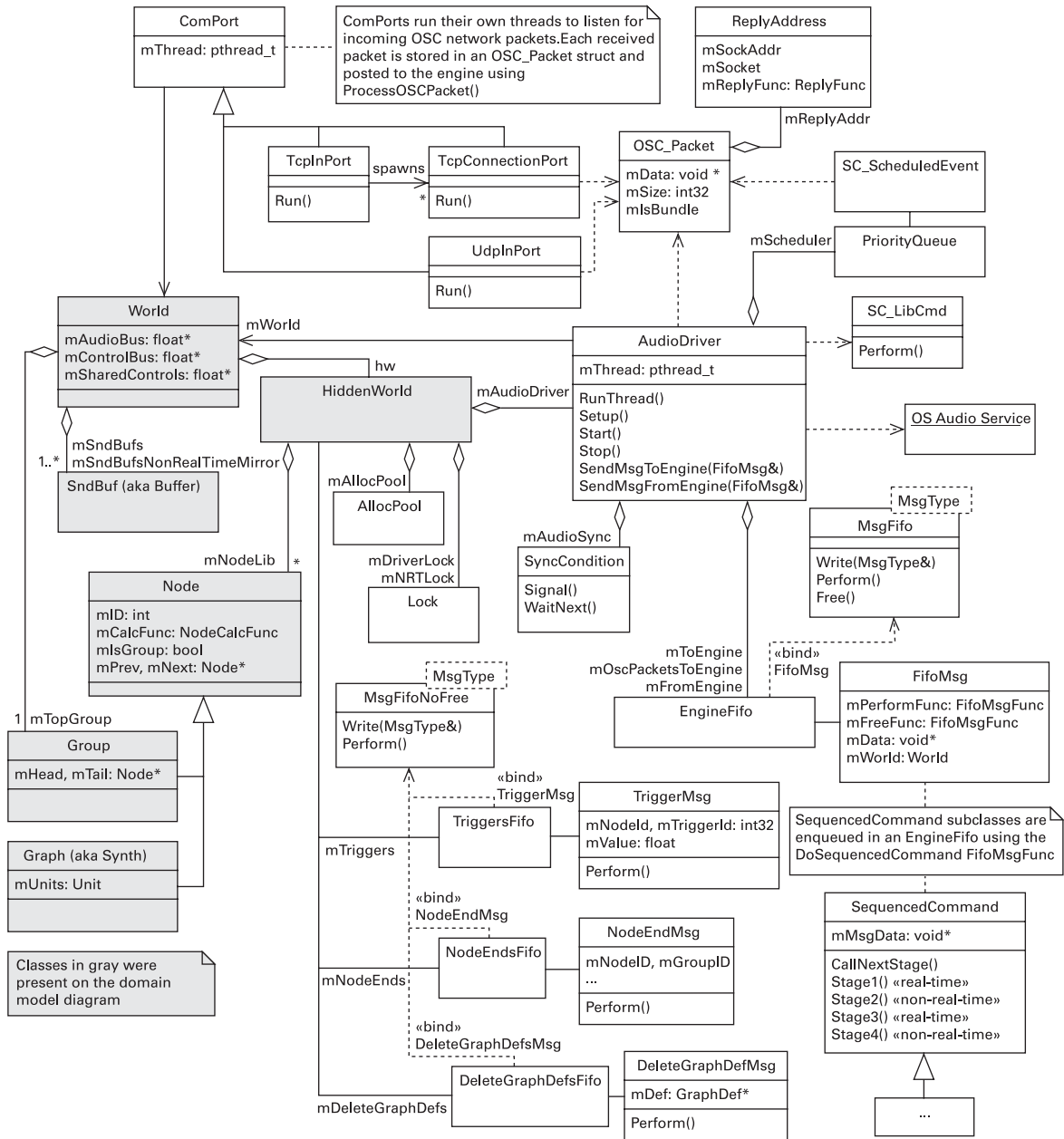


Figure 26.2 Real-time threading and messaging implementation structure.

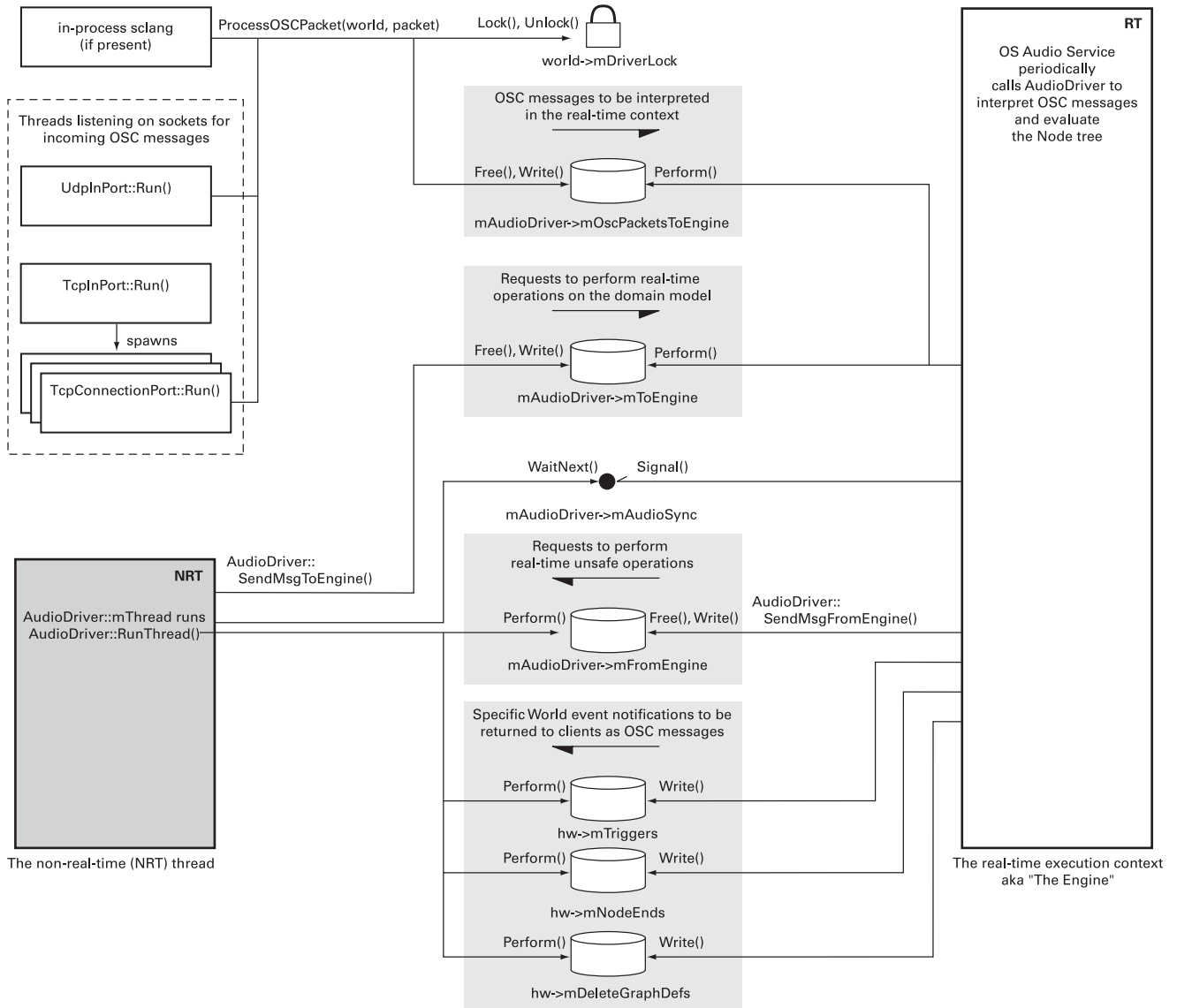


Figure 26.3 Real-time thread and queue instances and asynchronous message channels.

new Graphs, and so on. Whenever the Engine wants to perform a non-real-time safe operation, it encodes the operation in a `FifoMessage` instance and posts it to the non-real-time thread for execution via the `mFromEngine` queue. Results of such operations (if any) will be returned via the `mToEngine` queue. After processing messages from `mOscPacketsToEngine`, `mToEngine`, and any previously scheduled OSC messages in `mScheduler`, the Engine performs its audio duties by arranging for real-time audio data to be copied between OS buffers and `mWorld->mAudioBus` and evaluating the Node tree via `mWorld->mTopGroup`. When the Engine has completed filling the OS audio output buffers, it calls `Signal()` on `mAudioSync` and returns to the OS.

3. Before the server starts servicing OS audio requests, it creates a thread for executing real-time unsafe operations (the non-real-time or NRT thread). This thread waits on `mAudioSync` until it is signaled by the Engine. When the non-real-time thread wakes up, it calls `Free()` and `Perform()` on the `mFromEngine` queue to perform any non-real-time safe operations which the server has posted, then processes the `mTriggers`, `mNodeEnds`, and `mDeleteGraphDefs` queues. These queues contain notifications of server events. Performing the enqueued notification messages results in OSC messages being sent to clients referenced by `ReplyAddress`. After calling `Perform()` on all queues, the non-real-time thread returns to waiting on `mAudioSync` until it is next wakened by the Engine. Note that `mAudioSync` is used to ensure that the NRT thread will always wake up and process Engine requests in a timely manner. However, it may never sleep, or it may not process the queues on every Engine cycle if it is occupied with time-consuming operations. This is acceptable since the Engine assumes non-real-time operations will take as long as necessary.

The description above has painted the broad strokes of the server's real-time behavior. Zooming in to a finer level of detail reveals many interesting mechanisms which are worth the effort to explore. A number of these are discussed in the sections which follow.

26.3.2.1 Real-time memory pool allocator

Memory allocations performed in the real-time context, such as allocating memory for new Graph instances, are made using the `AllocPool` class. `AllocPool` is a reimplementation of Doug Lea's fast general-purpose memory allocator algorithm (Lea, 2000). The implementation allocates memory to clients from a large, preallocated chunk of system memory. Because `AllocPool` is invoked only by code running in the real-time context, it doesn't need to use locks or other mechanisms to protect its state from concurrent access and hence is real-time safe. This makes it possible for the server to perform many dynamic operations in the real-time thread without needing to defer to an NRT thread to allocate memory. That said, large allocations and other memory operations which are not time-critical are performed outside the real-time context. Memory allocated with an `AllocPool` must of course also be freed

into the same `AllocPool`, and in the same execution context, which requires some care to be taken. For example, `FifoMsg` instances posted by the Engine to the NRT thread with a payload allocated by `AllocPool` must ensure that the payload is always freed into `AllocPool` in the real-time execution context. This can be achieved using `MsgFifo::Free()`, which is described in the next section.

25.3.2.2 FIFO queue message passing

As already mentioned, `scsynth` uses FIFO queues for communicating between threads. The basic concept of a FIFO queue is that you push items on one end of the queue and pop them off the other end later, possibly in a different thread. A fixed-size queue can be implemented as a *circular buffer* (also known as a *ring buffer*) with a read pointer and a write pointer: new data are placed in the queue at the write pointer, which is then advanced; when the reader detects that the queue is not empty, data are read at the read pointer and the read pointer is advanced. If there's guaranteed to be only 1 reading thread and 1 writing thread, and you're careful about how the pointers are updated (and take care of atomicity and memory ordering issues) then it's possible to implement a thread-safe FIFO queue without needing to use any locks. This lock-free property makes the FIFO queue ideal for implementing real-time interthread communications in `scsynth`.

The queues which we are most concerned with here carry a payload of message objects between threads. This is an instance of the relatively well known *Command* design pattern (Gamma et al., 1995). The basic idea is to encode an operation to be performed as a class or struct, and then pass it off to some other part of the system for execution. In our case the *Command* is a struct containing data and a pair of function pointers, one for performing the operation and another for cleaning up. We will see later that `scsynth` also uses a variant of this scheme in which the *Command* is a C++ class with virtual functions for performing an operation in multiple stages. But for now, let's consider the basic mechanism, which involves posting `FifoMsg` instances to a queue of type `MsgFifo`.

Figure 26.2 shows that `mOscPacketsToEngine`, `mToEngine`, and `mFromEngine` queues carry `FifoMsg` objects. The code below shows the `FifoMsgFunc` type and the key fields of `FifoMsg`.

```
typedef void (*FifoMsgFunc)(struct FifoMsg*);

struct FifoMsg {
    ...
    FifoMsgFunc mPerformFunc;
    FifoMsgFunc mFreeFunc;
    void* mData;
    ...
};
```

To enqueue a message, the sender initializes a `FifoMsg` instance and passes it to `MsgFifo::Write()`. Each `FifoMsg` contains the function pointer members `mPerformFunc` and `mFreeFunc`. When the receiver calls `MsgFifo::Perform()`, the `mPerformFunc` of each enqueued message is called with a pointer to the message as a parameter. `MsgFifo` also maintains an additional internal pointer which keeps track of which messages have been performed by the receiver. When `MsgFifo::Free()` is called by the sending execution context, the `mFreeFunc` is invoked on each message whose `mPerformFunc` has already completed. In a moment we will see how this mechanism is used to free `SequencedCommand` objects allocated in the real-time context.

A separate `MsgFifoNoFree` class is provided for those FIFOs which don't require this freeing mechanism, such as `mTriggers`, `mNodeEnds`, and `mDeleteGraphDefs`. These queues carry specialized notification messages. The functionality of these queues could have been implemented by dynamically allocating payload data and sending it using `FifoMsg` instances; however, since `MsgFifo` and `MsgFifoNoFree` are templates parameterized by message type, it was probably considered more efficient to create separate specialized queues using message types large enough to hold all of the necessary data rather than invoking the allocator for each request.

The `FifoMsg` mechanism is used extensively in `scsynth`, not only for transporting OSC message packets to the real-time engine but also for arranging for the execution of real-time unsafe operations in the NRT thread. Many server operations are implemented by the `FifoMsgFuncs` defined in `SC_MiscCmds.cpp`. However, a number of operations need to perform a sequence of steps alternating between the real-time context and the NRT thread. For this, the basic `FifoMsg` mechanism is extended using the `SequencedCommand` class.

26.3.2.3 SequencedCommand

Unlike `FifoMsg`, which just stores two C function pointers, `SequencedCommand` is a C++ abstract base class with virtual functions for executing up to 4 stages of a process. Stage 1 and 3 execute in the real-time context, while stages 2 and 4 execute in the NRT context. The `Delete()` function is always called in the RT context, potentially providing a fifth stage of execution. `SequencedCommands` are used for operations which need to perform some of their processing in the NRT context. At the time of writing, all `SequencedCommand` subclasses were defined in `SC_SequencedCommand.cpp`. They are mostly concerned with the manipulation of `SndBufs` and `GraphDefs`. (See table 26.1 for a list of `SequencedCommands` defined at the time of writing.)

To provide a concrete example of the `SequencedCommand` mechanism, we turn to the `Help` file for `Buffer` (aka `SndBuf`), which reads: “Buffers are stored in a single global array indexed by integers beginning with zero. Buffers may be safely allocated, loaded and freed while synthesis is running, even while unit generators are using them.” Given that a `SndBuf`'s sample storage can be quite large, or contain

Table 26.1
Subclasses of `SequencedCommand` Defined in `SC_SequencedCommand.cpp`

Buffer Commands	<code>BufGenCmd</code> , <code>BufAllocCmd</code> , <code>BufFreeCmd</code> , <code>BufCloseCmd</code> , <code>BufZeroCmd</code> , <code>BufAllocReadCmd</code> , <code>BufReadCmd</code> , <code>SC_BufReadCommand</code> , <code>BufWriteCmd</code>
GraphDef Commands	<code>LoadSynthDefCmd</code> , <code>RecvSynthDefCmd</code> , <code>LoadSynthDefDirCmd</code>
Miscellaneous	<code>AudioQuitCmd</code> , <code>AudioStatusCmd</code> , <code>SyncCmd</code> , <code>NotifyCmd</code> , <code>SendFailureCmd</code> , <code>SendReplyCmd</code> , <code>AsyncPlugInCmd</code>

sample data read from disk, it is clear that it needs to be allocated and initialized in the NRT thread. We now describe how the `SequencedCommand` mechanism is used to implement this behavior.

To begin, it is important to note that the `SndBuf` class is a relatively lightweight data structure which mainly contains metadata such as the sample rate, channel count, and number of frames of the stored audio data. The actual sample data are stored in a dynamically allocated floating-point array pointed to by `SndBuf::data`. In the explanation which follows, we draw a distinction between instance data of `SndBuf` and the sample data array pointed to by `SndBuf::data`.

In contrast to the client-oriented worldview presented in the Help file, `World` actually maintains 2 separate arrays of `SndBuf` instances: `mSndBufs` and `mSndBufsNonRealTimeMirror`. Each is always in a consistent state but is accessed or modified only in its own context: `mSndBufs` in the RT context via `World_GetBuf()` and `mSndBufsNonRealTimeMirror` in the NRT thread via `World_GetNRTBuf()`. On each iteration the engine performs messages in `mToEngine` and then evaluates the Node tree to generate sound. Any changes to `mSndBufs` made when calling `mToEngine->Perform()` are picked up by dependent Units when their `UnitCalcFunc` is called.

The code may reallocate an existing `SndBuf`'s sample data array. It is important that the old sample data array is not freed until we can be certain no Unit is using it. This is achieved by deferring freeing the old sample data array until after the new one is installed into the RT context's `mSndBufs` array. This process is summarized in figure 26.4. The details of the individual steps are described below.

We now consider the steps performed at each stage of the execution of `BufAllocReadCmd`, a subclass of `SequencedCommand`, beginning with the arrival of an `OSC_Packet` in the real-time context. These stages are depicted in 4 sequence diagrams, figures 26.5 through 26.8. The exact function parameters have been simplified from those in the source code, and only the main code paths are indicated to aid understanding. The OSC message to request allocation of a Buffer filled with data from a sound file is as follows:

```
/b_allocRead bufnum path startFrame numFrames
```

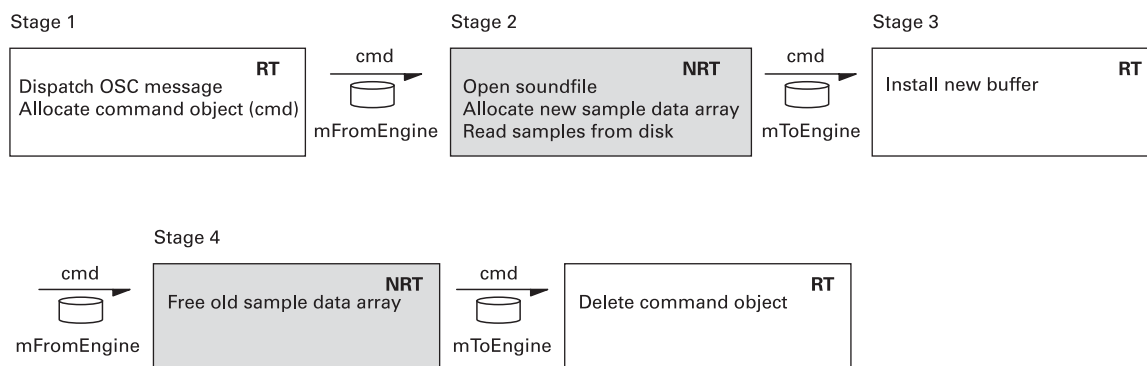


Figure 26.4

Overview of multithreaded processing of the `/b_allocRead` command.

Stage 1 (see figure 26.5): The real-time context processes an OSC packet containing the `/b_allocRead` message. The OSC dispatch mechanism looks up the correct function pointer to invoke from `gCmdLibrary`, in this case `meth_b_allocRead()`. `meth_b_allocRead()` calls `CallSequencedCommand()` to instantiate a new `BufAllocReadCmd` instance (a subclass of `SequencedCommand`) which we will call `cmd`. `CallSequencedCommand()` calls `cmd->Init()`, which unpacks the parameters from the OSC packet and then calls `cmd->CallNextStage()`, which in turn invokes `cmd->Stage1()`, which in the case of `BufAllocReadCmd` does nothing. It then enqueues `cmd` to the NRT thread, using `SendMessageFromEngine()` with `DoSequencedCommand` as the `FifoMsgFunc`.

Stage 2 (see figure 26.6): Some time later, the `mFromEngine` FIFO is processed in the NRT thread. The `FifoMsg` containing our `cmd` is processed, which results in `cmd->Stage2()` being called via `DoSequencedCommand()` and `cmd->CallNextStage()`. `cmd->Stage2()` does most of the work: first it calls `World_GetNRTBuf()`, which retrieves a pointer to the NRT copy of the `SndBuf` record for `cmd->mBufIndex`. Then it opens the sound file and seeks to the appropriate position. Assuming no errors have occurred, the pointer to the old sample data array is saved in `cmd->mFreeData` so it can be freed later. Then `allocBuf()` is called to update the `SndBuf` with the new file information and to allocate a new sample data array. The data are read from the file into the sample data array and the file is closed. A shallow copy of the NRT `SndBuf` is saved in `cmd->mSndBuf`. Finally, `cmd->CallNextStage()` enqueues the `cmd` with the real-time context.

Stage 3 (see figure 26.7): Similarly to stage 2, only this time in the real-time context, `cmd->Stage3()` is called via `DoSequencedCommand()` and `cmd->CallNextStage()`. A pointer to the *real-time* copy of the `SndBuf` for index `cmd->mBufIndex` is retrieved

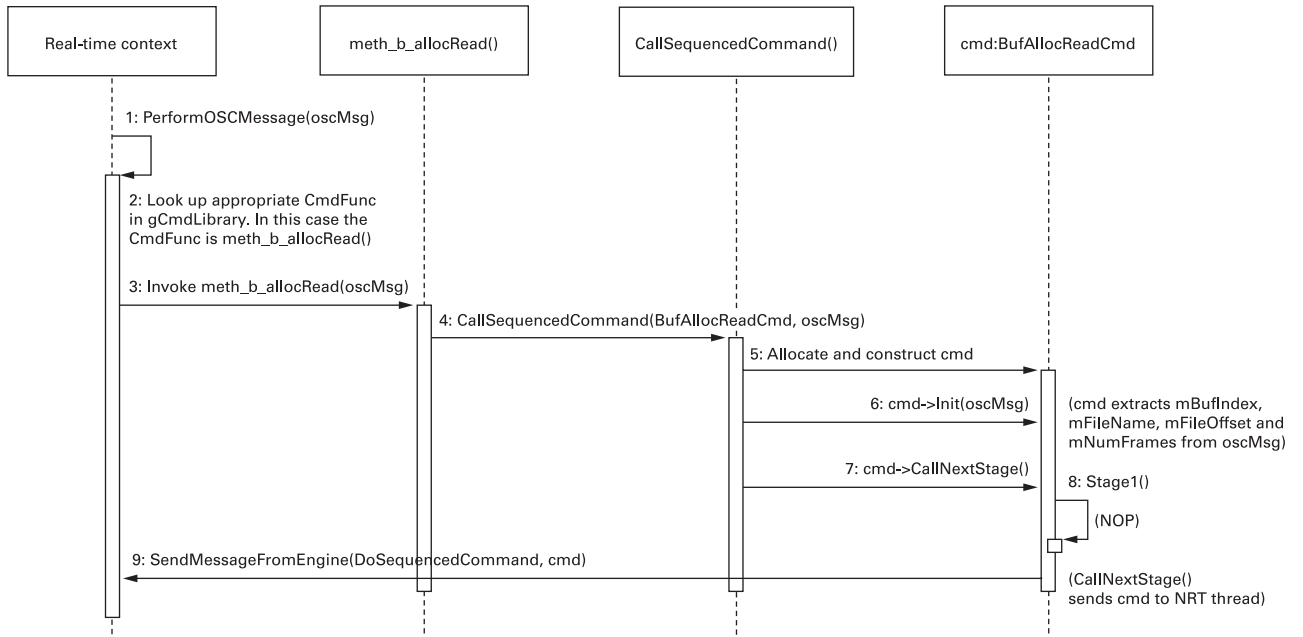


Figure 26.5
Stage 1 of processing the `/b_allocRead` command in the real-time context.

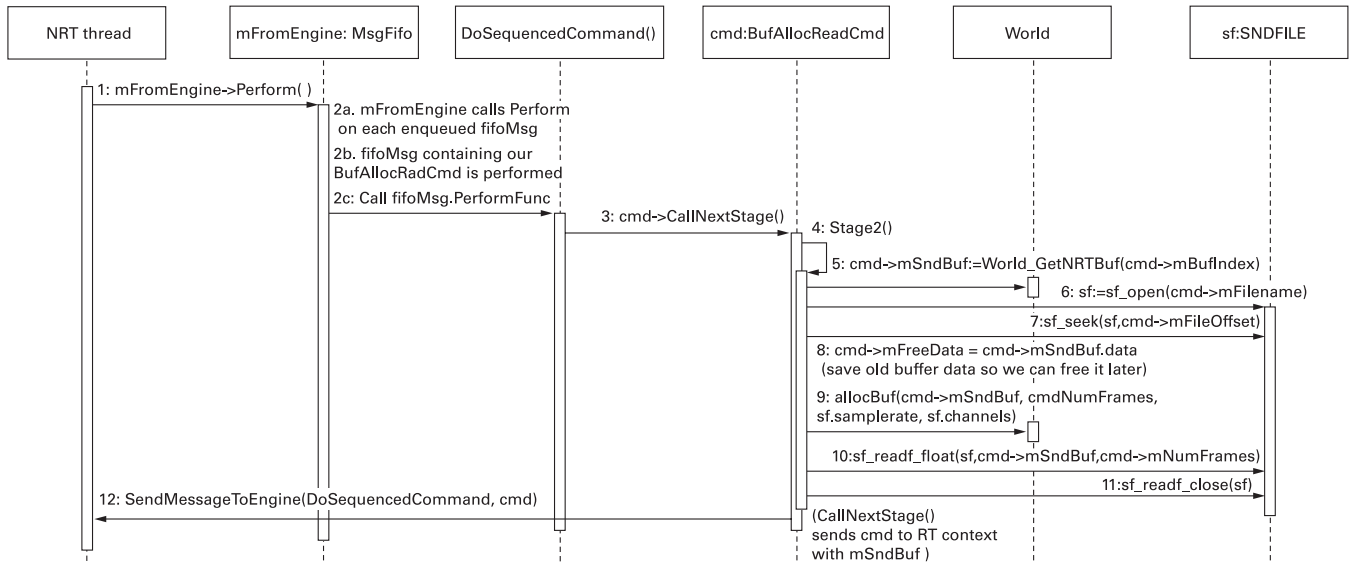


Figure 26.6
Stage 2 of processing the `/b_allocRead` command in the non-real-time (NRT) context.

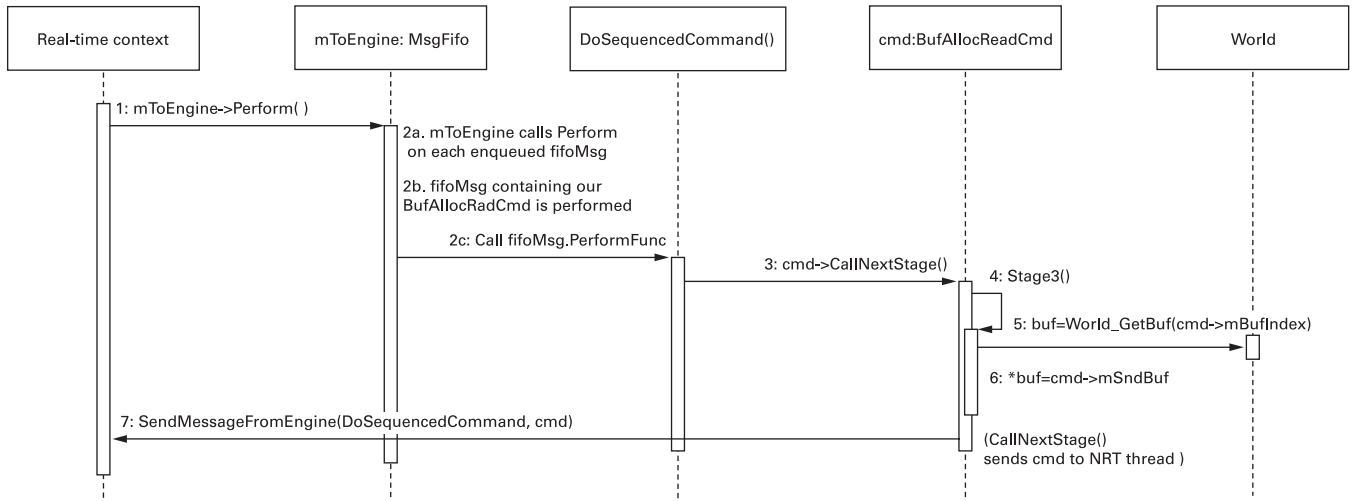


Figure 26.7

Stage 3 of processing the `/b_allocRead` command in the real-time context.

using `World_GetBuf(cmd->mBufIndex)`, and the `SndBuf` instance data initialized in stage 2 is shallow copied into it from `cmd->mSndBuf`. At this stage the sample data array which was allocated and loaded in stage 2 is now available to Units calling `World_GetBuf()`. `cmd` is then sent back to the non-real-time thread.

Stage 4 (see figure 26.8): Once again, back in the non-real-time thread, `cmd->Stage4()` is invoked, which frees the old sample data array which was stored into `cmd->mFreeData` in stage 2. Then the `SendDone()` routine is invoked, which sends an OSC notification message back to the client who initiated the Buffer allocation. Finally, `cmd` is enqueued back to the real-time context with the `FreeSequencedCommand FifoMsgFunc`, which will cause `cmd` to be freed, returning its memory to the real-time `AllocPool`.

26.3.2.4 Processing and dispatching OSC messages

The `ProcessOSCPacket()` function provides a mechanism for injecting OSC messages into the real-time context for execution. It makes use of `mDriverLock` to ensure that only 1 thread is writing to the `mOscPacketsToEngine` queue at any time (this could occur, for example, when multiple socket listeners are active). To inject an OSC packet using `ProcessOSCPacket()`, the caller allocates a memory block using `malloc()`, fills it with an OSC packet (for example, by reading from a network socket), and then calls `ProcessOSCPacket()`. `ProcessOSCPacket()` takes care of enqueueing the packet to the `mOscPacketsToEngine` queue and deleting packets, using `free()`, once they are no longer needed.

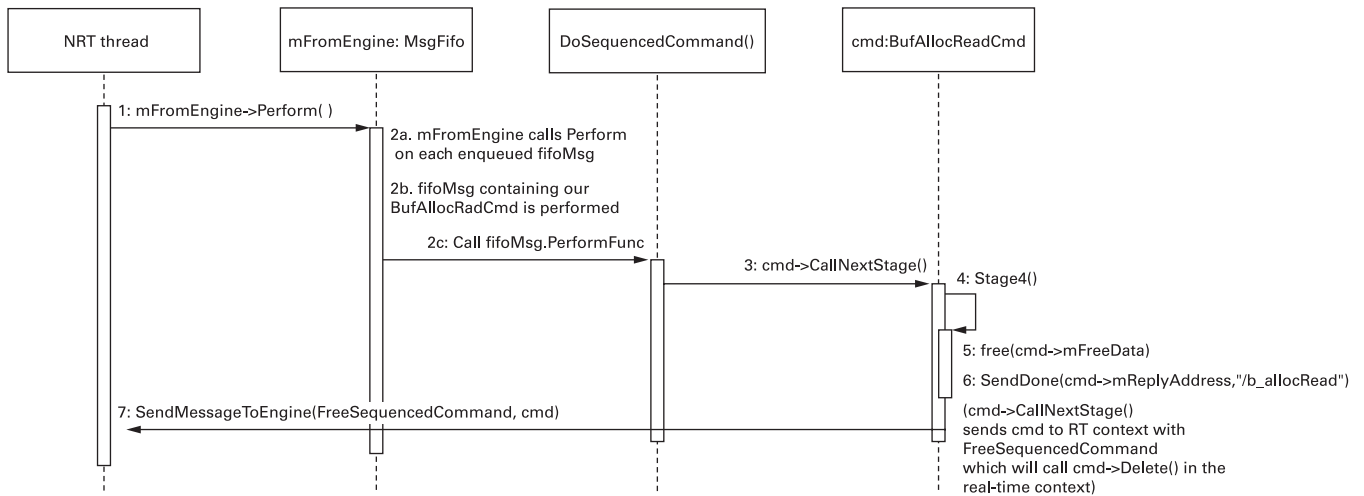


Figure 26.8

Stage 4 of processing the `/b_allocRead` command in the non-real-time (NRT) context.

Once the real-time context processes OSC packets, they are usually freed using the `MsgFifo` message-freing mechanism; however, packets whose time-stamp values are in the future are stored in the `mScheduler PriorityQueue` for later execution. Once a scheduled packet has been processed, it is sent to the NRT thread to be freed.

`scsynth` dispatches OSC commands by looking up the `SC_CommandFunc` associated with a given OSC address pattern. At startup `SC_MiscCmds.cpp` wraps these functions in `LibCmd` objects and stores them into both the `gCmdLib` hash table and `gCmdArray` array.

OSC commands sent to the server may be strings or special OSC messages with a 4-byte address pattern in which the low byte is an integer message index. Command strings are compatible with any OSC client, whereas the integer command indices are more efficient but don't strictly conform to the OSC specification. When integer command indices are received, `PerformOSCMessage()` looks up the appropriate `SC_CommandFunc` in the `gCmdArray` array; otherwise it consults the `gCmdLib` hash table.

The `mTriggers'`, `mNodeEnds'`, and `mDeleteGraphDefs'` FIFOs are used by the real-time context to enqueue notifications which are translated into OSC messages in the NRT thread and are sent to the appropriate reply address by invoking `ReplyAddress::mReplyFunc`.

26.3.2.5 Fixed-size data structures

In real-time systems a common way to avoid the potential real-time unsafe operation of reallocating memory (which may include the cost of making the allocation and of

copying all of the data) is simply to allocate a “large enough” block of memory in the first place and have operations fail if no more space is available. This fixed-size allocation strategy is adopted in a number of places in scsynth, including the size of

- FIFO queues which interconnect different threads
- `mAllocPool` (the real-time context’s memory allocator)
- The `mScheduler` priority queue for scheduling OSC packets into the future
- The `mNodeLib` hash table, which is used to map integer Node IDs to Node pointers.

In the case of `mNodeLib` the size of the table determines the maximum number of Nodes the server can accommodate and the speed of Node lookup as `mNodeLib` becomes full. The sizes of many of these fixed-size data structures are configurable in `WorldOptions` (in general, by command line parameters), the idea being that the default values are usually sufficient, but if your usage of scsynth causes any of the default limits to be exceeded, you can relaunch the server with larger sizes as necessary.

26.4 Low-Level Mechanisms

As may already be apparent, scsynth gains much of its power from efficient implementation mechanisms. Some of these fall into the category of low-bounded complexity methods which contribute to the real-time capabilities of the server, while others are more like clever optimizations which help the server to run faster. Of course the whole server is implemented efficiently, so looking at the source code will reveal many more optimizations than can be discussed here; however, a number of those which I have found interesting are briefly noted below. As always, consult the source code for more details.

- The `Str4` string data type consists of a string of 32-bit integers, each containing 4 chars. Aside from being the same format that OSC uses, the implementation improves the efficiency of comparison and other string operations by being able to process 4 chars at once.
- Hash tables in scsynth are implemented using open addressing with linear probing for collision resolution. Although these tables don’t guarantee constant time performance in the worst case, when combined with a good hashing function (Wang, 2007) they typically provide close to constant performance so long as they don’t get too full.
- One optimization to hashing used in a number of places in the source code is that the hash value for each item (such as a Node) is cached in the item. This improves performance when resolving collisions during item lookup.
- The `World` uses a “touched” mechanism which Units and the `AudioDriver` can use to determine whether audio or control buses have been filled during a control cycle:

World maintains the `mBufCounter`, which is incremented at each control cycle. When a Unit writes to a bus, it sets the corresponding touched field (for example, in the `mAudioBusTouched` array for audio buses) to `mBufCounter`. Readers can then check the touched field to determine whether the bus contains data from the current control cycle. If not, the data doesn't need to be copied and zeros can be used instead.

- Delay lines typically output zeros until the delay time reaches the first input sample. One way to handle this is to zero the internal delay storage when the delay is created or reset. The delay unit generators in `scsynth` (see `DelayUGens.cpp`) avoid this time-consuming (and hence real-time unsafe) operation by using a separate `UnitCalcFunc` during the startup phase. For example, `BufDelayN_next_z()` outputs zeros for the first `bufSamples` samples, at which point the `UnitCalcFunc` is switched to `BufDelayN_next()`, which outputs the usual delayed samples.

- For rate-polymorphic units, the dynamic nature of `UnitCalcFuncs` is used to select functions specialized to the rate type of the Unit's parameters. For example, `BinaryOpUGens.cpp` defines `UnitCalcFuncs` which implement all binary operations in separate versions for each rate type. For example, there are separate functions for adding an audio vector to a constant, `add_ai()`, and adding 2 audio vectors, `add_aa()`. When the binary-op Unit constructor `BinaryOpUGen_Ctor()` is called, it calls `ChooseNormalFunc()` to select among the available `UnitCalcFuncs` based on the rate of its inputs.

This concludes our little journey through the wonderful gem that is `scsynth`. I invite you to explore the source code yourself; it has much to offer, and it's free!

References

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Design*. Reading, MA: Addison-Wesley.

Lea, D. 2000. "A Memory Allocator," <<http://g.oswego.edu/dl/html/malloc.html>> (accessed January 9, 2008).

McCartney, J. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal*, 26(4): 61–68.

Wang, T. 1997. "Integer Hash Function," <<http://www.concentric.net/~Ttwang/tech/inthash.htm>> (accessed January 9, 2008).