Supporting Device Drivers In The SquawkVM

Alex Garthwaite

May 16, 2005

1 Introduction

This document describes the basic design for supporting device drivers in the SquawkVM that are written in the Java™ programming language. This design, due to Nik Shaylor, has three components:

- message-service communication channels,
- device- and interrupt-specific message structures, and
- support for kernel and user contexts so that drivers may be executed asynchronously with respect to application code.

Here, we outline these facilities, how they have been extended, and how they might be modified in future systems.

2 Message Queues

Message-service channels are a form of channel IO; they are distinguished by using the "msg:" prefix in URLs. They offer a simple mechanism whereby a client may open a URL, send some data as part of a request, and receive a response from the server thread. This kind of exchange is illustrated in figure 1.

Supporting this form of exchange are five global message queues:

- **toServerMessages.** messages destined to be handled by server connections,
- **toClientMessages.** messages destined to be received by clients.
- **toServerWaiters.** server threads blocked waiting for data,
- **toClientWaiters.** client threads blocked waiting for data, and
- **messageEvents.** events intended to wake threads blocked on IO requests.

Data is communicated between client and server threads via the first two queues; threads attempting to read data from not-yet-generated messages are handled with the last three queues. In all fives cases, the same basic message structure—the `MessageStruct`—is used. These are formed into queues as shown in figure 2.

The use of `MessageStructs` for thread scheduling are relatively straight-forward: the message, itself, is used as an identifier for the event to be posted when input becomes available. The use of `MessageStructs` for exchanging data is more interesting (cf. figure 3). The `MessageStruct` provides a `next` field for linking

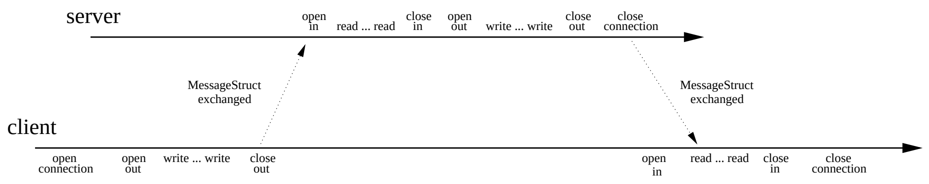

Figure 1: Example illustrating how clients and servers interact via the message-service infrastructure.

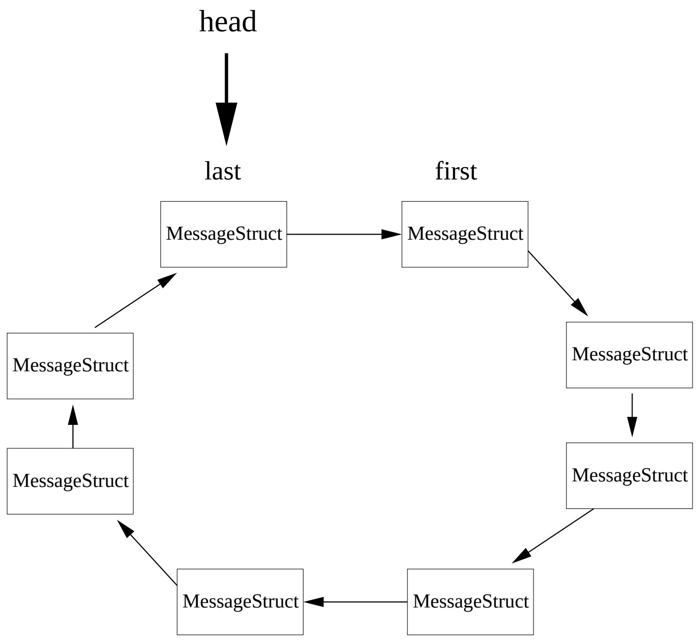

Figure 2: Circular list of `MessageStructs` forming a message queue.

`MessageStructs` together, a `status` field for notifying the recipient of changes in the status of the connection, a `data` field linked to one or more `MessageBuffers` containing data, and a `name` or `key` field naming the URL identifying the channel to which the `MessageStruct` belongs. The `MessageBuffers`, themselves, break up the data being exchanged into 128-byte packets each with a `next` field, a `position` field used in reading data from each `MessageBuffer`, a `byte count` field identifying how much data is available in a `MessageBuffer`, and a `data` field containing the data being exchanged. Data is read in FIFO order from the `MessageBuffers` linked off of a given `MessageStruct`, and the amount available for reading from a given `MessageStruct` is the sum of the byte counts of its `MessageBuffers`.

Clients and servers share an asymmetric relationship. A client may send at most one `MessageStruct` as part of its request to a service URL. Once it begins reading data, however, its output to the service connection is closed and it may only read

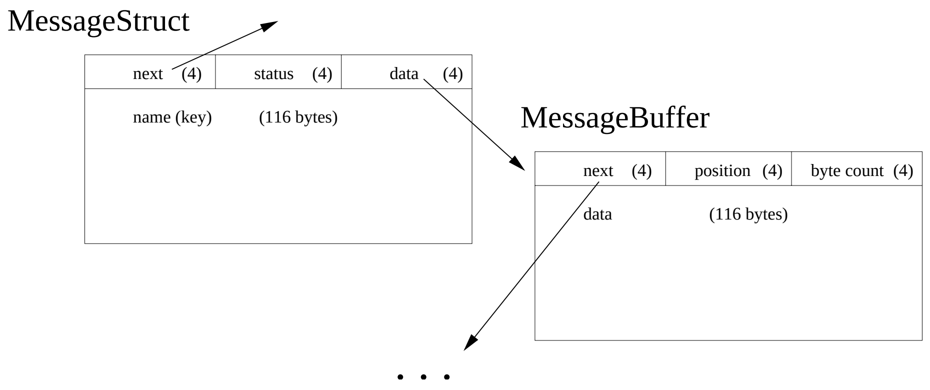

Figure 3: A message packet formed from a `MessageStruct` together with zero or more `MessageBuffers`.

data thereafter. Data is read from the current `MessageStruct` and attempts to read an exhausted `MessageStruct` result in an `IOException` being thrown. To allow clients to receive a stream of `MessageStructs` over time from a single service request, the client may issue a `reset` operation on its input: this action drops the existing data and allows a `read` operation to load in a new `MessageStruct`. On the server side, a server thread cannot reset its input but it can send multiple `MessageStructs` back to the client by performing `flush` operations. In all cases, packets are only exchanged when a `flush` or `close` operation is performed.

All of these connections are implemented via the `MessageInputStream` and `MessageOutputStream` classes. To obtain the differing sets of behavior, each server or client connection makes use of a protocol class: `ServerProtocol` for server connections, and `ClientProtocol` for client connections.

In many cases, it is better if the server thread does not block. For this purpose, Nik introduced a polling mechanism via the VM.getNextServerConnectionHandler operation. This enables the server thread to query the `toServerMessages` queues for any `msg:` URLs that have been registered via the `VM.addServerConnectionHandler` operation.

One point of interest: message queues are persistent. Clients and servers can open any valid `msg:` URL and post messages and there is no guarantee that these structures will ever be processed and recycled.

3 Single Context

The simplest approach to support message services is to run both the server and client connections in a single context. In this case, each side is run in separate threads and scheduled by the SquawkVM. This, for example, is the mode that Nik used to do preliminary testing of the message-service infrastructure. The downside to this approach is that devices and device-related events such as interrupts must be handled in native code and the results polled periodically by the JVM threads. The upside is that it re-

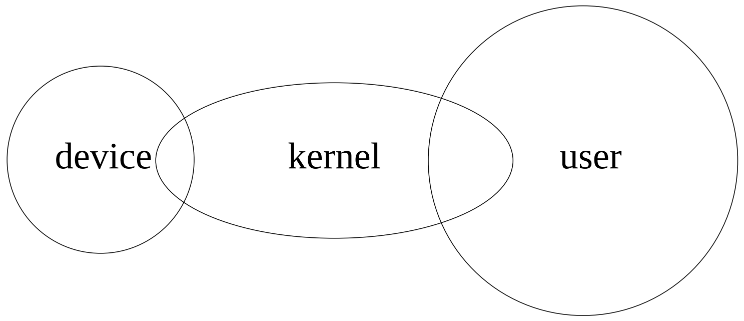

Figure 4: Overall structure of the two-context approach.

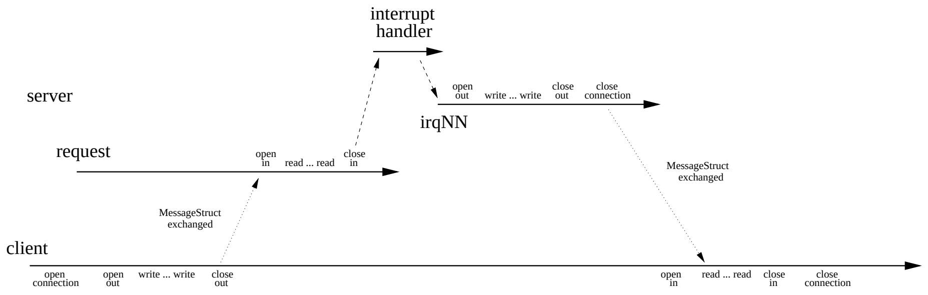

Figure 5: Example illustrating how device drivers extend the message-service infrastructure.

quires no change to the original design of the SquawkVM with a single context and heap.

4 Two Contexts

A more complicated approach is to support two contexts: a user context that supports the existing SquawkVM model and a kernel context that allows drivers written in the Java programming language to handle asynchronous events and communicate with devices. The basic model can be seen in figure 4; the extended exchange of messages between application, driver, and interrupt handlers can be seen in figure 5. The idea is that the kernel manages its own event loop that processes messages from devices delivered via interrupts and communicates these results to client threads executing in the user context. To make this work, each of the contexts has its own heap, its own set of threads, . Communication between the two contexts is only through data structures malloc'd or statically allocated such as those provided by the message services. In this section, we describe some of the design decisions that have gone into this approach.

4.1 Transitioning into and out of the kernel context

One question is: how is the kernel context entered? There are three mechanisms. The first, and simplest, is during the initialization of the JVM, the kernel context is initialized and its main routine invoked. Once it established its initial set of drivers, it returns control to the user context by calling `VM.pause()`. In fact, this routine is intended to be the only place in which the kernel thread blocks. As such, it relies on the `VM.getNextServerConnection` operation to poll for messages to be processed before handing control back to the user context.

Whenever a client thread sends a message to a server connection, the code issues an interrupt causing a special interrupt handler to be entered. This handler checks if the kernel thread will have any work to perform and whether it is already active. If it is not active and there are messages to be processed, execution resumes in the kernel context. To make sure these conditions hold on exit from the kernel, the checks are always performed with interrupts disabled.

In a similar manner, whenever a device-related interrupt is received, its own handler is entered. This handler makes use of a statically allocated `MessageStruct` specific to that interrupt and posts this message (via a `postMessage` operation) to the appropriate device URL (typically beginning with a string such as "irq"). It also makes a check whether the kernel context should be be resumed and does so if not.

Notification of the presence of new data to either side occurs in two phases. First, when a new `MessageStruct` is placed on a message queue, the corresponding waiters queue is checked for matches. Those entries identifying the same message URL are removed from the waiters queue and placed on the `messageEvents` queue. Later, when the `reschedule` operation is performed, these events are popped and the identified threads are returned to the runnable queue.

A final wrinkle concerns the handling of events on the `messageEvents` queue. While not true of the current sample configurations, this global queue could potentially have events pending for both kernel and user threads. Because these events, delivered during rescheduling decisions such as calls to `yield()`, should only be delivered in the appropriate context, we have chosen to reuse the status field for `MessageStructs` in the `toServerWaiters` and `toClientWaiters` queues to record the originating context for each entry. We then use this context information in deciding which events to deliver in each context.

4.2 Device-related data structures

To support the posting of device-specific messages on message queues, we follow Nik's design in preallocating a set of memory for these structures as well as interrupt-specific statistics that we may want to keep. The model is that one case of a particular interrupt may be handled at a time; its handling places its designated `MessageStruct` on the appropriate device message queue. When the kernel thread processes this message, it uses a special hook in the `close` operation called `checkForDeviceInterrupt` that resets the state of the interrupt and allows its `MessageStruct` to be reused. Clearly, the current implementation wastes some space and could be made smaller, but it is intended for pedagogical purposes only.

4.3 Interrupts and handlers

The interrupt mechanism interacts with the kernel context in interesting ways. As we have previously mentioned, various checks relating to whether control should enter, or remain in, the kernel context require that signals be masked. Here, we outline some of the other ways in which interrupt-handling is used.

Second, interrupts are being employed for several purposes in the current design:

- to synchronously transition from the user context into the kernel context,
- to save the machine state and allow native code to execute, and
- to signal that a device has completed some operation.

Further, by appropriately masking and unmasking (sets of) signals, we can control when and where these operations occur. One might wish to write, for example, much of the interrupt handlers, themselves, in the Java programming language but this would require either significant rearchitecting of the SquawkVM to support true concurrency, or the support for more contexts than the current two.

By employing only two contexts, we have made some other compromises. While we are able to respond to asynchronously delivered interrupts in a more timely fashion than if we had a single context, and we are able to implement (almost all of) the functionality of the device drivers in the Java programming language, we are still in no way offering real-time guarantees in terms of servicing interrupts. This added functionality may not be needed, but it would also require some ability to support true concurrency or the presence of more contexts each with their own heaps.

4.4 Exchanging data

An interesting question arises: how does a driver written in the Java programming language exchange data with an interrupt handler? There are at least six ways that this can be achieved.

Native accessors. The simplest approach is to provide native methods for getting and retrieving values from statically allocated or malloc'd data structures. For example, such a mechanism is currently used to retrieve statistics about particular interrupt values.

Raw addresses to data outside of kernel heap. One variant of this approach is return the raw address of such a structure and then use driver code written in the Java programming language to use unsafe operations to access it.

Raw addresses to data in the kernel heap. A separate variant is to provide one or more addresses of heap objects to the native code so that it can write into these structures. This, for example, is how ExampleIODriver is written.

Exported class fields. A fourth variant is to export these structures (in either direction) by exporting them as symbols in classes loaded in the JVM. This, however, has all of the drawbacks of the approaches that employ explicit native methods for registering or exchanging data. Such choices must be built into a given platform and are neither easily changed nor portable across platforms.

Interrupt MessageStructs. One more flexible approach is to export the addresses of device interrupt `MessageStructs` to the drivers written in the Java programming language and to then use unsafe operations to manipulate the `MessageBuffer`(s) associated with these structures. This has the added advantage of using a common data-exchange mechanism for the message services whether these exchanges occur between drivers and client threads or between drivers and devices.

How these `MessageBuffers` are allocated to device interrupt `MessageStructs` is a matter of choice. They may be statically allocated and assigned when the interrupt is set up or they may be malloc'd (and later freed) for each use. The latter approach has the added advantage of allowing `MessageBuffers` associated with an interrupt's `MessageStruct` to be transferred directly to a `MessageStruct` being used to communicate with some client thread. For example, such an exchange would be ideal in transferring an array of data directly from the device's buffer to the client thread without forcing the driver to recopy that data into a `MessageBuffer`. Support for this transfer, however, would require adding methods to `MessageOutputStream`.

Interrupt URL key fields. A final mechanism for exchanging information would be to make use of extensions to the current naming or key field in each interrupt's `MessageStruct`. Similar to tagging WWW URL's with additional information, this field could be used to communicate short (typically a few dozen bytes) sequences of information without allocating additional memory. It would also require the exporting of the addresses of these fields for this purpose.

The existing sample drivers have tended to use the first N approaches. However, to reduce the need to register native methods, to reduce the need to copy data in some instances, and to simplify the overall design of device drivers, the mechanisms that either reuse the device `MessageStructs` or their name/key fields to pass data are to be preferred.

4.5 Interaction with embedded VMs

One wrinkle for instances of the SquawkVM running with an embedded J2SE JVM such as the HotSpot™ Java Virtual Machine is that these instances rely on the embedded JVM to support an event loop handling, for example, keyboard IO. The problem is that this event-loop mechanism was not designed to handle the possibility that all client threads are suspended when an asynchronous response from an interrupt—for example, an alarm notification—is delivered.

To make this work, the kernel must, on transitioning control back to the user context, connect to the embedded JVM via JNI. Because JNI associates a unique `JNIEnv` structure with each native thread and the kernel and user contexts share a native thread, we employ a helper native thread whose sole function is to be awoken by the kernel context and to send a `notifyAll` message to the embedded JVM so that it wakes and reschedules any threads that are now runnable.

Fortunately, this issue should not affect the SunSpots in any way.

5 Open Issues

There are currently a number of open issues. These include:

- controlling how message services are established,
- providing a registry of messaage services,
- improving client-side error messages and feedback,
- purging information in transit for a message service,
- extending the notion of connection beyond a simple URL,
- supporting the unloading of device drivers,
- improving portability of device drivers,
- moving more C code into the Java programming language, and
- extending the message-service infrastructure to support either the trace or debug facilities.

Here, we offer a short description of each of these and why they might be important in a production system.

Establishing message services. The original message-service implementation was intended to support either the client and server threads executing in a single context or in two. It allows server connections to be opened by any thread. This means, however, that any thread may do so, and any thread may also register the service URL with `addServerConnectionHandler()`. While only a class in `java.lang` may access `getNextServerConnectionHandler()` and this provides some security, it probably is not a good idea that any thread may initiate such a service.

In addition, it might be useful in some future implementation to allow for multiple lists of registered services so that different threads in different contexts could wait on a particular set of message URLs depending on where in the JVM it is best to place particular services.

Providing a registry. In the current implementation, there is no mechanism for a client thread to query which services are available. Further, it has been left as an open design question whether these URLs should be high-level (identifying, for example, a timer or radio device), or load-level (identifying, for example, a particular port behind the SPI controller). Such a global registry would be very useful for clients to query.

Improving client-side feedback. We added support for a mechanism to report status information back to either the kernel or user contexts. This enables us, for example, to report that a client has passed the wrong set of parameters for a particular service request. The reporting mechanism is to raise an `IOException` for the client thread when it attempts to read a message.

This reporting mechanism could be used in a number of other ways. For example, if a client thread opens a connection to a URL for which there is no service, it is allowed to do so. In fact, it is further allowed to send a request on that connection's output stream and to wait for a response. It would be better in such cases if the original attempt to open the connection had resulted in an appropriate exception being thrown. This also would not be difficult in the presence of a global registry of services. Likewise, the out-of-band reporting mechanism could aid in the purging of message queues of messages when some client threads are still awaiting input.

Another use would be to be able to identify whether a `MessageStruct` sent to the client is part of a stream of packets or is *terminal*. The idea would be to provide feedback to the client thread as to whether it can flush the state for the current packet and expect to read another. A flush of a terminal packet would ensure that any successive read would result in an `IOException` identifying the EOF condition. Similarly, reporting message numbers for each of the streaming packets may be useful to the client threads, particularly if there are multiple consuming threads.

Purging information in transit for a message service. There are times when it would be useful to purge all messages relating to some device. For example, if one changes the timing characteristics of a timer alarm, one might want to eliminate all in-flight timer-related messages. If no client thread is waiting for input, one could simply eliminate the pending messages from the queues. However, if a client thread is known to be waiting for input (because it is blocked in a `read` request), it may be useful to use the status-reporting mechanism to report to the client thread that the message has been purged.

There is also no way to guarantee that some client will consume a set of messages in the message queues. This is true even if the client explicitly closes a connection explicitly without reading the data. In these cases, it would be useful to reduce the potential for floating garbage by being able to eliminate these pending messages.

Qualifying individual connections. One worry is that either side of a message service, there is no security as to what thread may open either a `serverConnection` or a `streamConnection` for any `msg:` URL. All interactions simply match on this string, and so, any thread—even one that opens the same URL multiple times—will see the same server- or client-side data stream. It would be better if each `open()` provided a unique counter or identifier with each stream and that this unique identifier were part of the message exchange. Such an extension would not be difficult to add and would also make the ability to purge and/or close a particular client-side connection to a service possible.

Unloading device drivers. Currently, we have a simple device driver for loading additional ones. However, we have no mechanism for unloading devices. Adding this would require taking a number of steps, some that we have already noted are not supported:

- purging message queues and setting appropriate client-side events,
- unregistering the message-service URL from the `serverConnection` list,
- performing any clean-up on the supported device.

Improving portability of device drivers. The existing sample device drivers communicate information via native methods. These native methods limit the ability to load new drivers into the kernel if those drivers require additional native support. Much of these constraints can be eliminated by using `MessageStructs` and `MessageBuffers` to exchange both data and status information. However, this will not help drivers that must take some specific action to interact with a device.

A secondary aspect of making device drivers and their support more portable is extending the mechanism for specifying how the driver-manager is initialized. Currently, one can specify options for the kernel context by using the '-K' prefix. It would be good to also be able to specify the class containing the driver-manager's `main()` routine rather than assuming this class is `JavaDriverManager`.

Expressing more code in the Java programming language. Some of the existing code—`Message.java`, for example—manipulates raw addresses via the Unsafe interface. Currently, these instances are limited to accesses to `MessageBuffers`. It may be good to move some of code for manipulating the `MessageStructs`, themselves, into Java. As we noted above, this would allow us to assume fewer native methods for accessing status information and communicating information to and from devices.

Support for tracing and debugging. No work has been done to make either of these facilities interact with a kernel context. The tracing facility, for example, would have been extremely useful in debugging some of the problems encountered in transitions between the two contexts. Support for debugging would also be useful but presents a whole new set of issues:

- how are signals masked/delivered in the context of a debugging session?
- can we suspend a thread in the kernel context?
- how is control followed when switching between the two contexts?

It may be simpler to not allow the debugger to suspend or introspect on the kernel side so that, for example, the handling of asynchronous interrupts may be handled in a timely fashion.