icras

Magic Internet Kit

Programmer’s Guide

April 24, 2000

Josh and Ed's Excellent Internet Kit

Lyrics and electric banjo by Josh Carter. Drums and synths by Ed Satterthwaite. Backing vocals by Zarko Draganic, Dean Yu, and C.J. Silverio. Tour management by Mark "The Red" Harlan.

Get out the banjo and let's boogie to this

Icras, Inc.

955 Benecia Avenue	Tel.:	408 530 2900
Sunnyvale, CA 94086 USA	E-mail:	info@icras.com
	Fax:	408 530 2950
	URL:	http://www.icras.com/

Table of Contents

1

Introduction

Real connectivity made real easy

The Magic Internet Kit is a complete development kit for creating communicating Magic Cap applications. The heart of the Magic Internet Kit is an easy-to-use object framework which provides:

- TCP/IP support for writing full-featured Internet/Intranet applications.
- Supporting protocols for TCP/IP such as PPP, Ethernet, and DNS.
- Serial communications over a DataRover 840's built-in modem and serial port.
- Ability to add additional hardware drivers without needing to change or rebuild client applications.

From the first steps of creating a communicating application to maintaining the code later, the Magic Internet Kit makes your work easier and your development cycle faster. From the very beginning, you can use one of several templates provided with the kit to get you started right where you want to be. These templates range from a basic Finger client to a ready-to-go, multi-threaded terminal package.

While the templates get you started quickly, the real power of the Magic Internet Kit lies in its robust and flexible programming interfaces. This framework is arranged to provide you with a single set of methods that you can call to create any type of communications stream supported by the kit.

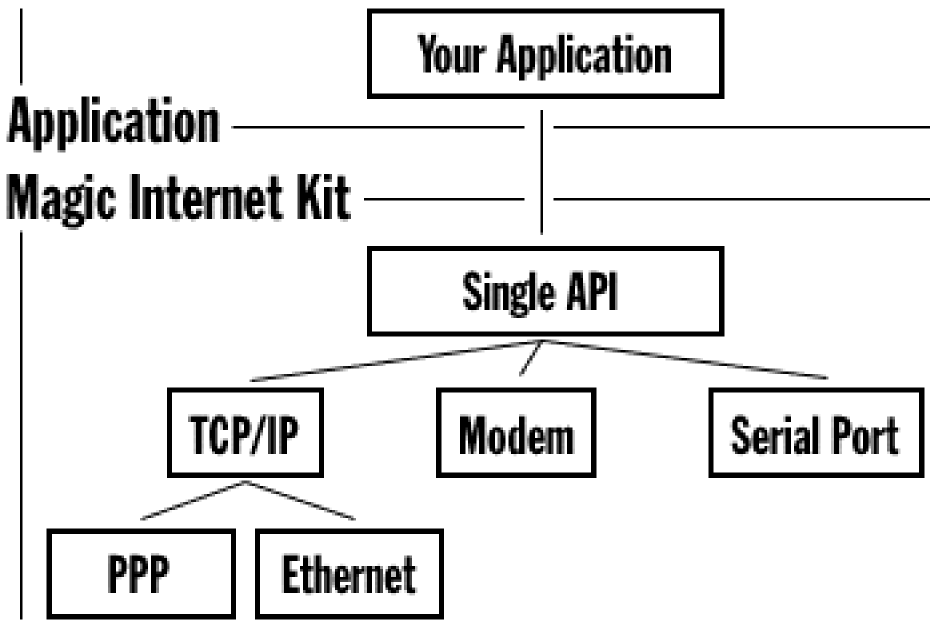

Figure 1 Magic Internet Kit layout

As illustrated, many means of communication are available on the bottom end, and the application can access all of them through one API. The Magic Internet Kit API will be discussed in the *Connections* chapter of this guide.

What you should already know

The *Magic Internet Kit Programmer's Guide* assumes you are familiar with the basics of TCP/IP and related protocols. If you need an introduction, I recommend *Internetworking with TCP/IP* by Douglas Comer or *TCP Illustrated* by Richard Stevens. If you don't want to write TCP/IP applications, then you don't need to worry; the Magic Internet Kit also works directly with the modem or serial port.

This document also assumes you are familiar with Magic Cap development. If you need to find out more about the basics of writing a Magic Cap application, see the Magic Cap *Package Development Guide* included with your software development kit.

About this document

The *Magic Internet Kit Programmer's Guide* starts with a discussion of the templates available for creating a package, and then proceeds to discuss the high-level APIs for communications. Later chapters dive into detail about TCP/IP and multithreaded application code. This section will briefly describe the topic of each chapter.

Creating a New Package

This chapter describes the process of creating a new communicating package. Topics include the template applications provided by the Magic Internet Kit and how to clone a template for your own application.

Connections

This chapter discusses the centerpiece of the Magic Internet Kit: the **Connection** class. First it describes the API provided by Connection, and then describes specific subclasses of Connection, including **InternetConnection**.

Streams

This chapter describes how to use Magic Cap communications streams. The **Stream** class is used for reading and writing data, so topics include the methods used to read, write, and catch errors.

Multithreading with Actors

Multithreading allows your application to run code "in the background," which is very important with time-consuming communications tasks. This chapter tells you how to use multithreading, and demonstrates the concepts with the CujoTerm template.

2

Creating a New Package

Templates

The first step in creating a new package is to clone an existing template. These templates are functional Magic Cap packages in themselves, but they are designed to be starting points for your own application. This chapter describes the templates provided in the Magic Internet Kit and how to use them.

Templates provided in the kit

Templates are provided to give you a good starting point for a new application. There are several interface elements that many communicating packages may want to use – for example a choice box that lets the user pick an Internet provider – so the Magic Internet Kit lets you start with varying levels of functionality built in. This section will describe each of the included templates.

Note: These templates are contained in your software development kit's *Samples* folder.

Terminal (a.k.a. CujoTerm)

This is the classic terminal package. CujoTerm gives you a simple terminal that you can connect to a remote host over whatever connections means the kit supports. CujoTerm's **TerminalField** class is a **TextField** subclass which displays text from the communication stream, accepts typing and sends data to the stream, and even lets the user drop text coupons that will be sent to the stream.

CujoTerm also uses a **ProviderChoiceBox**, a subclass of **ChoiceBox**, which allows the user to choose an Internet service provider (ISP) for the connection. ProviderChoiceBox is defined by the core Internet Kit framework since it's very handy, so you can use this class even if you don't start from the CujoTerm template. Internet providers are set up in Magic Cap's Internet Center, so your package does not need to worry about this feature; you can get to the information using ProviderChoiceBox or your own code.

The primary feature of CujoTerm is its architecture. This terminal, while basic in user-level features, has an advanced multi-threaded structure behind it that can support multiple simultaneous TCP/IP connections. The two classes implementing this are the **CommsActor** and **CommsManager**. The code executing on its own thread is part of the CommsActor, and the CommsManager is used to easily create and destroy CommsActors. The concepts behind these classes will be discussed in greater detail in the *Multithreaded Communications* chapter of this document.

CujoTerm is a good template to use for testing code or investigating protocols. For example, if you wanted to write a package that queried and responded to a database, you could use CujoTerm to make sure that the commands you are sending to the server are actually doing the right thing. Additionally, since CujoTerm works out of the box, you can use it as a good sample package to learn from.

Finger

The Finger template package is a basic finger client as defined by RFC 742. Its user interface is very sparse, but like CujoTerm it include s the handy ProviderChoiceBox class for picking an Internet provider. Finger is a good clean slate to start a new package from if you don't anticipate using many of CujoTerm's goodies.

Cloning a template

Cloning a template is just like cloning any other package. Refer to your development tools guide for the appropriate procedure.

Where to go from here

Now that you're set up with a clone of your preferred template, it's time to make it do some real work. The next chapter, *Connections*, shows you how to connect to a remote host, use the connection, and then destroy it when you're done.

3

Connections

The heart of the Magic Internet Kit is its easy-to-use object framework for connecting your application to the outside world. This act can take several forms, from serial communications to TCP/IP over a variety of data links. This chapter describes the programming interfaces your package uses to create, use, and destroy links to remote hosts.

The Connection class

The **Connection** class defines the shared programming interface for creating all types of communication streams. Subclasses of Connection are defined for each type of communications means supported by the kit; for example **InternetConnection** is used for TCP/IP-based communications.

Connection subclasses serve two purposes: they contain the data needed to establish their data stream, and they implement the API defined to by Connection to create and destroy their streams. Like other Magic Cap objects, connection data is stored in fields that are accessed via attributes. The exact fields are dependent on the type of connection, so templates like CujoTerm use **AttributeText** objects and variants thereof to provide a user interface for filling in data. This style allows the package to have an appropriate user interface for the connection type, but the interface is separate from the application code.

This section will discuss the API defined by the Connection class, and later sections will discuss details of each of the Connection subclasses.

Source Code Note: The Connection class definition can be found in the Connection:Connection.cdef file inside your Magic Internet Kit folder.

Methods of Connection

Connection defines three methods: CanCreateStream, CreateStream, and DestroyStream. We'll cover what each one does in detail.

CanCreateStream

```
operation CanCreateStream(): Boolean;
```

This method is called by client code to ask if the connection could potentially create a stream. For example, if this Connection subclass communicates using the modem, CanCreateStream would check to see if the phone line is plugged in, make sure the user has set up a dialing location, and check that the modem is not already in use.

Be aware that CanCreateStream does not guarantee that a connection can be made, but only that a connection *cannot* be made. Many factors affecting the connection cannot be determined until the attempt is actually made; for example, if a remote server you are trying to contact is offline, you can't know that until you try connecting to it. CanCreateStream is still useful, however, because it allows you to make a quick check for common problems before launching threads or other communications setup code.

CreateStream

```
operation CreateStream(): Stream;
```

CreateStream is the method that tells a Connection to create a stream to its remote host. If the attempt succeeds, this method will return a reference to the stream it created. The stream returned from CreateStream is always a subclass of **CommunicationStream**, for example the **Modem** class for serial modem connections, or the **TCPStream** class for TCP/IP connections.

If the connection attempt fails, CreateStream will throw a **CommsException**. Error handling with CommsExceptions is slightly different than many other exceptions in Magic Cap, so this topic will be discussed shortly in the following *CreateStream error handling* section.

Note: With the InternetConnection class, you can call CreateStream repeatedly to get multiple streams over the same data link.

DestroyStream

```
operation DestroyStream(stream: Stream);
```

DestroyStream is the inverse of CreateStream; when a stream is no longer needed, this method is used to destroy it in whatever manner is appropriate. This method will also destroy any objects or buffers that were allocated by CreateStream. For example, if CreateStream creates a TCPStream object, DestroyStream will be sure to destroy that object and its local buffers. Never call Magic Cap's **Destroy** method on a stream returned by CreateStream; always use the kit's DestroyStream method since it knows how to deal with each type of stream appropriately.

CreateStream error handling

As a general rule with communications, connecting to a remote host requires many things to all work together, so there are many places for things to go wrong. Errors in Magic Cap are typically handled with **exceptions** that get thrown when the error occurs and are then caught by application code designed to handle that error. The exception itself is usually an indexical that serves as a unique identifier for a type of error. For more information on handling these types of exceptions, see the Magic Cap *Package Development Guide*.

Given the multitude of possible errors that could occur when creating a communication stream, an application using typical Magic Cap exceptions would have to catch almost a dozen separate exceptions. The code for setting up the error handling could be pages long! The Magic Internet Kit takes a slightly different but much cleaner approach by using **heavyweight exceptions**.

A heavyweight exception is similar to any other exception in how it is used, but in this case the exception is not an error code but rather a real, live object. In the case of CreateStream, the object that is thrown is a **CommsException** or one of its subclasses. Code calling CreateStream, therefore, can catch all of the possible CommsExceptions by catching them by their class. The method that does this is called, quite reasonably, **CatchByClass**. Here is an example of how to use CatchByClass to catch all CommsException errors:

```
Reference exception = CatchByClass(CommsException_);

if (exception != nilObject)
{
    /* we caught a CommsException object */
    return false; /* your error handling code goes here */
}

stream = CreateStream(iMyConnection);

Commit(); /* CommsException_ */
```

Once you have caught a CommsException object, you can find out which exception it is by comparing it to the ones listed in the Connection:CommsException.odef file included in your Magic Internet Kit folder. For example:

```
if ((exception = CatchByClass(CommsException_)) != nilObject)
{
    /* we caught a CommsException object */
    if (exception == ieCannotConnectHardware)
    {
        /* can't connect the hardware to the remote host */
        return false;
    }
    else if /* etc... */
}
```

The CujoTerm template uses this approach and announces error messages to the user for each exception.

Additionally, errors are grouped into subclasses of CommsException, for example all three of the DNS-related errors are of the DNSCommsException class. This gives you an extra level of granularity if you want it. For the case of DNS, you can check to see if an exception was any of the DNS errors with the following code:

```
if ((exception = CatchByClass(CommsException_)) != nilObject)
{
    /* we caught a CommsException object */
    if (Implements(exception, DNSCommsException_))
    {
        /* a DNS-related error occured */
        return false;
    }
    else if /* etc... */
}
```

Furthermore, this feature means that you can CatchByClass on particular subclasses of CommsException if you want to have different handlers for them. In practice this last technique might not be useful since all CommsException errors are fatal for a given connection attempt, but it might come in handy.

Connection subclasses in the Magic Internet Kit

The Magic Internet Kit includes three subclasses of Connection: **InternetConnection** for creating TCP/IP streams, **ModemConnection** for creating dial-up serial modem streams, and **SerialPortConnection** for streams using the DataRover 840’s built-in serial port.

InternetConnection

InternetConnection is used to create and destroy TCP/IP streams. This is often a daunting task on many platforms, but InternetConnection takes care of all the details for you – your application only has to deal with the three methods defined by Connection. Thanks to the Internet Center, filling in the fields of an InternetConnection object is also a snap. The user fills in all of the required information for their Internet Service Provider (ISP) in the Internet Center, and then the application can easily use that information. First we’ll discuss filling in InternetConnection’s fields, and then how that data is used.

InternetConnection attributes

```
attribute   ServiceChoice:  InternetServiceChoice;
attribute   ServiceInfo:    InternetServiceInfo;
attribute   HostName:       Text;
attribute   HostPort:       Unsigned;
```

InternetConnection defines two key attributes: ServiceChoice and ServiceInfo. These are filled in with **InternetServiceChoice** and **InternetServiceInfo** objects. The first class, InternetServiceChoice, defines a service provider and contains the relevant data needed for a connection. This includes a **Means** object which holds information about the hardware driver, the ISP’s name, and other interesting things. Fortunately, your application does not need to create or set up these objects; they are created by the Internet Center when the user sets up an ISP. The easiest way to fill in this field is to use the Magic Internet Kit’s **ProviderChoiceBox** class. This class is used by both the Finger and CujoTerm templates, and it provides a list of the ISPs that the user has set up in the Internet Center. It then sets the iCurrentServiceChoice indexical to the matching InternetServiceChoice object. Simply put an instance of ProviderChoiceBox in your package and set the ServiceChoice attribute of your InternetConnection objects to iCurrentServiceChoice.

The second class, InternetServiceInfo, specifies the remote host name and port that you want to connect to. Setting a destination host name and port are very common tasks, so InternetConnection helps out by defining its own **HostName** and **HostPort** attributes. Getting and setting these attributes with an InternetConnection object will actually get and set fields within its InternetServiceInfo object, but your application does not notice the difference. Furthermore, creating a new InternetConnection object will automatically create a matching InternetServiceInfo object for it to use.

Note: The HostName attribute of InternetConnection can be either a symbolic host name (e.g. www.datarover.com) or an IP number (e.g. 204.188.97.7). If the host name is symbolic, InternetConnection will use Domain Name Resolution to look up the name automatically.

The easiest way to set up an InternetConnection is to use **AttributeText** and **UnsignedAttributeText** objects that point to the HostName and HostPort attributes. The first class, AttributeText, serves as the data store for a text field and fills in a text attribute of its target object with the field’s contents. The second class, UnsignedAttributeText, is a subclass of AttributeText defined by the Magic Internet Kit for targeting an Unsigned attribute. See the CujoTerm sample for using these two classes.

Source Code Note: Examples of AttributeText and UnsignedAttributeText can be found in the CujoTerm:InternetSetup.odef file.

Details of InternetConnection

InternetConnection hides the complexity of TCP/IP connections under the Connection class’s methods. CanCreateStream checks to make sure that its ServiceChoice and ServiceInfo objects have enough data for the connection, and also makes sure the hardware is available. One very useful feature in Magic Cap is that TCP/IP is part of the system itself, so multiple applications can transparently share a data link to the service provider.

CreateStream also uses Magic Cap’s ability for multiple applications to share a data link. If a data link to a given service provider does not exist, CreateStream will create one automatically. If a data link is already up and running – even if it was created by completely separate application – CreateStream will use the existing link. CreateStream will automatically look up symbolic host names if needed, so your application does not need to worry about name resolution (DNS).

DestroyStream is just as intelligent as CreateStream. If CreateStream starts up a new data link, DestroyStream will take it down as long as there are no other users. In fact, if another application starts up a stream on the data link created by your application, DestroyStream will leave the link in place for the other application. It will then be taken down automatically when the other application finishes.

ModemConnection

```
attribute   SerialServer:   CommunicationStream;
attribute   PhoneNumber:    Text;
```

ModemConnection is used for serial dial-up access to remote modems. This class is not used for PPP connections – you would use InternetConnection for that – but rather for raw serial links over a phone line. I recommend building applications using internet protocols if at all possible, but sometimes a direct dial-up link is the only way to access a legacy system.

Both ModemConnection and SerialPortConnection (discussed below) inherit from a common class named **SerialConnection**. SerialConnection defines a SerialServer attribute which directs the connection to specific hardware. In most cases, a ModemConnection's serial server will be iModem, the built-in modem on a DataRover 840. Additional hardware can be supported with add-on driver packages.

The PhoneNumber attribute is specified by the **HasPhoneNumber** mixin class which ModemConnection inherits from. The format of these phone numbers is the country-specific dialing prefix (+1 in the United States) followed by a tab, and then the number with area code. Any other dialing prefixes and the option of dialing the area code are determined by settings in the user's dialing location.

Note: If you want to dial a number explicitly, prefix the number with a double-quote; e.g. *"123* will dial exactly 123 regardless of the user's dialing location setup.

SerialPortConnection

```
attribute   SerialServer:   CommunicationStream;
attribute   BaudRate:       Unsigned;
```

SerialPortConnection is used to access the DataRover 840's built-in serial port. As before, I recommend using internet protocols instead of direct serial links – for example, running SLIP on the serial line instead – but sometimes a legacy system will only support direct serial links.

The SerialServer attribute of most SerialPortConnection objects will be iSerialAServer, the built-in serial port. The BaudRate attribute should be set to any rate supported by the port. The DataRover 840’s port can handle 300bps to 38,400bps and most speeds inbetween.

Note: Keep in mind that both ModemConnection and SerialPortConnection do not ensure reliable data transfer. Data may get garbled or dropped. If the data is critical, your application must provide error detection and correction! Using a reliable, error correcting protocol like TCP/IP is recommended if possible.

I’m connected. Now what?

Now that you know about creating and destroying streams with Connection objects, and how to fill in the fields of these objects, it’s time to look at the object returned by CreateStream: the stream. Streams are the topic of the next chapter.

4

Streams

Subclasses of **Stream** are used for sending data between Magic Cap and a remote host. There are two essential methods used with streams: **Read** and **Write**. There are other handy methods for finding out how many bytes are waiting to be read, or writing the contents of a null-terminated c-string, and we will discuss several of these.

Essential methods of Stream

Using Magic Cap streams is straightforward. **CountReadPending** returns the number of bytes available to read, **Read** reads the bytes, and **Write** writes bytes.

CountReadPending

```
operation CountReadPending(): Unsigned, noFail;
```

This method is used to find out how many bytes are in the stream's local buffer and are therefore available for immediate reading. Reading is a synchronous operation, so code that does not want to block while reading should always call CountReadPending first to check how many bytes are available.

Read

```
operation Read(buffer: Pointer; count: Unsigned): Unsigned, noFail;
```

This method is used to read a number of bytes from a stream into a buffer. If the number of bytes requested by Read are not yet available – for example the remote host has not sent them – Read will block until either all the bytes are received or an error occurs. The return value is the number of bytes that were read, and comparing this value to the number of bytes requested lets the application know if an error occurred; if fewer bytes were returned than requested, then an error occurred but Read still returned as much data as it could get.

One useful tactic when waiting for data is to block on a one character read and then read any other pending data. For example, the following code is used in the CujoTerm template application:

```
char  buffer[255];
ulong count = Read(stream, &buffer[0], 1);

if (count != 1)
{
    /* error case */
    Fail(iServerAborted);
}
else /* count == 1; no errors */
{
    count = CountReadPending(stream);

    if (count != 0)
    {
        /* fill the rest of the buffer */
        if (count > 254) count = 254;
        Read(stream, &buffer[1], count);
    }

    /* Remember that we already read one byte up above! */
    count++;

    HandleBytes(client, (Pointer)buffer, count);
}
```

In this case, the one character read will block until either data is available or an error occurs, for example the remote host closed the stream. When Read returns, we check to make sure that Read returned a character, and if not an exception is thrown. If the character was read okay, we fetch the rest of the pending bytes, up to 255 total, and process them.

Write

```
operation Write(buffer: Pointer; count: Unsigned), noFail;
```

Write, as one can imagine, is used to write data to a stream. With TCP streams, write will immediately return after placing the bytes into TCP's outgoing buffer. If the stream was closed for some reason, or some other error occurs, Write will instead throw an iServerAborted exception. As a result, all calls to Write should be prepared to catch iServerAborted exceptions and handle the error case.

Other useful methods

The stream class defines several additional methods for your convenience. All of these methods are based on the essential three methods discussed above, so error handling tactics are identical; methods that Read data should check return values, and methods that Write data should catch iServerAborted exceptions.

ReadUntil

```
operation ReadUntil(buffer: Pointer; endChar: UnsignedByte; maxLen:
       Unsigned; var numRead: Unsigned; VAR numinBuf: Unsigned): Boolean,
       noFail;
```

ReadUntil is just like Read, except that is used to read until a specified character is found, or until a specified number of characters is read. This method can be very useful for reading data one line at a time; your code can ReadUntil the EOL character very easily. The boolean return value determines if the character was found. If the return value is true, the character was found. This character will not be in the buffer, so the *numInBuf parameter will be one less than *numRead. If the return value is false, ReadUntil read until the maxLength parameter was reached but did not find the character, or ReadUntil encountered an error. To tell which problem caused ReadUntil to return false, compare *numRead to maxLength. If these values are the same, ReadUntil did not find the character, but if *numRead is less than maxLength, and error occurred while reading from the stream.

WriteLiteral

```
operation WriteLiteral(dataToWrite: Literal);
```

WriteLiteral is used to write a null-terminated c-style string to a stream. It will write everything in the string except for the null terminator.

WriteTextAsASCII, WriteTextAsUnicode

```
operation WriteTextAsASCII(text: HasText);
operation WriteTextAsUnicode(text: HasText);
```

These two methods write **Text** objects to a stream. The first method, WriteTextAsASCII, writes the contents of a Text object as 8-bit characters and will replace all non-ASCII characters with '?.' The second method, WriteTextAsUnicode, writes the text object as 16-bit unicode characters.

Synchronous stream issues

All stream methods are synchronous, meaning they will not return until they have either finished their task or have encountered an error. This detail may not be very important for writing on some streams – e.g. TCP buffers small writes and returns quickly – but it is very important for reading. The Read method will block until all bytes requested are actually read, so if the remote host is slow sending data, your application must make provisions for not making the user interface halt while waiting.

There are two means of ensuring that your application does not block user interaction. The first is to periodically check CountReadPending until all data that you want can be read, and then call Read knowing that the data is available in the stream's local buffer. The second, and by far most preferable way, is to do all blocking communications work on a separate thread. With this tactic, user interaction takes place on its own thread, so your communications thread can block and the user will not notice; they can go on using other parts of your application or Magic Cap. Multithreading is the topic of our next chapter.

5

Multithreading with Actors

Magic Cap is a multi-threading platform, and threads are very handy for communications, so this document will briefly describe how to use them. As you may have noticed with the Finger template, blocking all user interaction while dialing the phone or waiting for a remote server is, at best, very annoying for the user. Instead of executing time consuming code on the thread handling user interaction, applications should create their own threads for these tasks.

In Magic Cap, a thread is called an **Actor**. Applications wishing to create their own actors must subclass the Actor class and override its **Main** method to make it do what they want. This section will briefly cover how to use actors, but the Magic Cap *Package Development Guide* provides a more complete discussion of this topic. We recommend that you read that chapter as you have time.

Actor concepts

Like everything in Magic Cap, an actor is an object. The base Actor class does not do anything when you create it, but rather it is meant to be subclassed. The first method that you should override is Main. Main is the heart of the actor; when the actor is created, Main is executed. When Main returns, the actor is destroyed.

Magic Cap uses cooperative multitasking with its actors, so each actor should give time to other actors. This is done by calling **RunNext** on the scheduler, referenced by its class number **Scheduler_**. RunNext tells the scheduler to run the next waiting actor while keeping the current one in the queue.

Some methods you might call will automatically call RunNext if they are going to take a while to return. For example, if you call Read on a TCPStream object, and the Read cannot be immediately satisfied because all the bytes requested are not available, Read will call RunNext to let other actors do their stuff.

Creating an actor

Actors are created using the **NewTransient** method. Here's an example:

```
newActor = NewTransient(CommsActor_, nil);
```

This code will create a new CommsActor class. The second nil parameter is for parameters that one might want to pass for the new actor, and passing nil tells Magic Cap to use the default parameters. If you want to pass in parameters – for example the size of the execution stack that the actor should have – you can do so just like for any other object. Here's an example of setting the stack size manually:

```
NewActorParameters newActorParams;
ZeroMem(&newActorParams, sizeof(NewActorParameters));

newActorParams.stackSize = 0x2000; /* default is 0x1000 (4K) */

newActor = NewTransient(CommsActor_, &newActorParams);
```

Once the actor is created, it will be ready to run in the scheduler. Keep in mind that the code in your actor's Main method will not start executing until the scheduler switches to the actor. This will happen the next time that you, or the system, calls RunNext.

Using an actor

As mentioned above, all the real work of an actor is performed in the Main method. Here's a sample main method:

```
Method void
MyActor_Main(Reference self, Pointer UNUSED(params))
{
    while (true)
    {
        DoSomeStuff(self);

        if (SomeFatalErrorOccurred(self))
        {
            return;
        }
    }
}
```

There is one essential caveat to using actors which you need to be aware of: code running on anything but the User Actor cannot call methods that draw on the screen. This means that code in the above MyActor_Main method cannot move viewable objects, call RedrawNow, or otherwise change stuff in the user interface. If you need to modify viewables or draw on the screen, you must use a special method called RunSoon to execute a function on the User Actor.

Note: There is one exception to the rule of not messing with stuff on screen from outside the User Actor: announcements. You can always call the **Announce** method regardless of the current actor since it will automatically use RunSoon if needed.

There is one more caveat to using actors in Magic Cap: an actor is a transient object, so it may or may not get destroyed when the device is power cycled. If Magic Cap decides to clean up transient memory while powering up or down, the actor will be destroyed, but otherwise it will stay around and start executing when the device powers up again. As a result, special care must be taken to ensure that the state of an actor does not get confused if the power is turned off and on. This is particularly important with communications, of course, because any data links will get shut down when the power is turned off.

One option for managing transient actors is overriding the **ResetClass** method of an object. ResetClass gets called when the device is powered on, so you can use this method to hunt down any actors that need to be managed and do whatever is appropriate. For communications this usually means destroying the actor. Another option is mixing in the **WantsPowerEvents** class and overriding **PoweringOn**.

Source Code Note: The Magic Internet Kit's CujoTerm template illustrates how to override ResetClass and destroy remaining actors at power-up time. See the CommsManager class in CujoTerm:CommsManager.[cdef/cpp].

Destroying an actor

Code can destroy an actor using Magic Cap's **DestroyActor** method. If the code wanting to kill the actor is running on the actor in question, though, it should instead force the actor's Main method to return. Returning from Main will destroy the actor automatically.

Moving between actors

Magic Cap provides mechanisms for code executing on one actor to talk to other actors, so we'll briefly cover those mechanisms here. There are a few different ways to manage multiple threads, including semaphores, cross-actor exception throwing, and the RunSoon method mentioned above. RunSoon is not covered in this document at the time.

Semaphores

A **Semaphore** object is very handy for controlling access to a resource. Magic Cap's semaphores are more intelligent than the traditional semaphore in operating system theory, in fact they behave almost like monitors. Semaphores provide automatic queueing and dequeueing as needed to control access to a single resource from multiple threads, so there is no need to poll a semaphore.

There are two key methods for using a Semaphore: **Access** and **Release**. Access is used to hold down the semaphore. If the semaphore being accessed is not already held down by code on another actor, Access will return immediately. If the semaphore is already in use, though, Access will block until someone else releases the semaphore using Release. Release tells the semaphore that you're done messing with it, and it will then wake up the next actor in the queue that wants to access the semaphore.

Cross-actor exceptions

If you're not already familiar with exception handling in Magic Cap, you should read the "Handling Exceptions" chapter of the Magic Cap *Package Developers Guide* for a good introduction. This section briefly describes how to use exceptions with actors.

Every actor has its own exception stack, so an exception thrown on one actor using Fail cannot be caught from another actor. This is very useful for localizing exception handling code; for example an iServerAborted exception thrown on a communicating application's comms actor won't be caught accidentally by some other actor.

If code executing on one actor wants to throw an exception for a different actor to catch, it should use the **FailSoon** method. FailSoon will cause a specified exception to get thrown on the other actor the next time that the actor comes up in the scheduler. If the target actor is not ready to run, e.g. it is blocked, the actor will be awakened and the exception thrown.

Source Code Note: See CommsManager_DestroyCommsActor in the CujoTerm template for an example of using FailSoon.

Actors in the Magic Internet Kit

If all of this multithreading stuff looks intimidating, don't fret – the Magic Internet Kit comes to the rescue. The CujoTerm template makes heavy use of actors. All connecting and reading is performed on its CommsActor object. Additionally, there is a CommsManager class that is used to create and destroy CommsActor objects. You can freely modify these to suit your own needs.

Source Code Note: See the CommsActor.[cdef/cpp] and CommsManager.[cdef/cpp] files in CujoTerm for its use of actors.