General Magic™

Magic Internet Kit

Programmer's Guide

September 6, 1996

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 and C.J. Silverio. Tour management by Mark "The Red" Harlan.

Get out the banjo and let's boogie to this

General Magic

420 North Mary Avenue	Telephone:	408 774 4000
Sunnyvale	Fax:	408 774 4010
California 94086 USA	E-mail:	dev-info@genmagic.com
	URL:	http://www.genmagic.com/

Patent Pending

Table of Contents

Introduction

Real connectivity made real easy

The Magic Internet Kit is a complete development kit for creating Magic Cap applications that communicate in a variety of ways. The heart of the Magic Internet Kit is a powerful yet easy-to-use object framework that provides:

- TCP/IP support for writing full-featured Internet/Intranet applications.
- Supporting protocols for TCP/IP such as PPP, with PAP and CHAP authentication, and DNS for resolving host names.
- Serial communications over a communicator's built-in modem and MagicBus port.
- Native PPP and serial support for external modems, including Metricom's Ricochet wireless modem.

From the first steps of creating a communicating application to maintaining the code later, the Magic Internet Kit makes your work easy and your development cycle fast. From the very beginning, you can use MIK's automated package creation helper tool to create an application with exactly the functionality that you want.

In addition to being able to pick and choose the functionality for your new application, you can also use one of several templates provided with MIK to get you started right where you want to be. These templates range from an "empty" package with no user interface to a ready-to-go terminal package.

While the handy package creation tools help you get 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 you want.

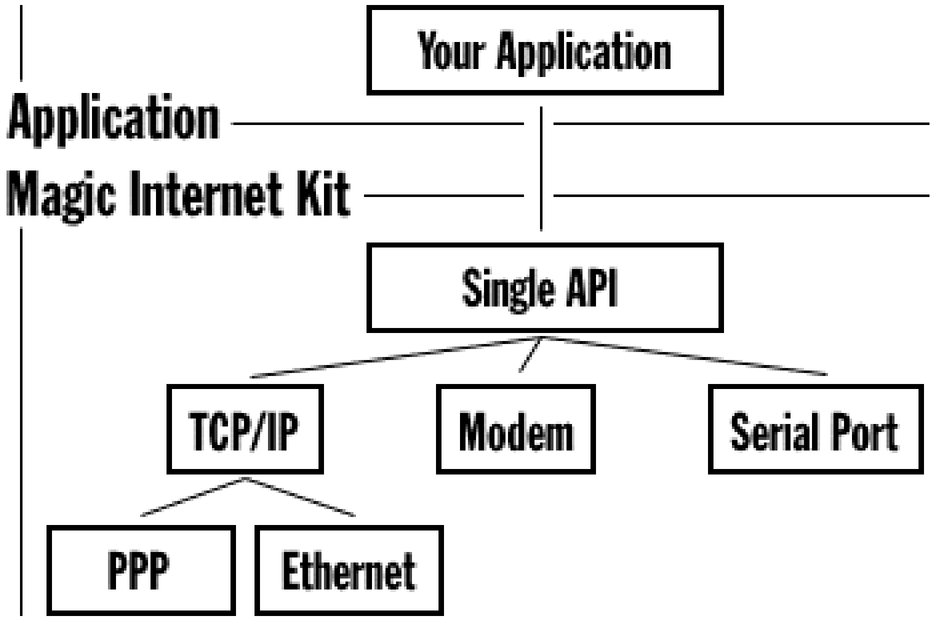


Figure 1 Magic Internet Kit layout

Any of the blocks in the above figure below the API can be dropped out or replaced with no effects on other blocks beside it. For example, if your application does not need serial modem or MagicBus support, these components can be left out with no adverse effects on TCP/IP. Thanks to this modular design, the Kit's framework is easily updateable when new modules are developed or current ones are enhanced. Additionally, all of the Internet Kit, with the exception of the low-level TCP/IP parts, is provided as fully documented and commented source code, so you can modify it if your application requires it.

What you should already know

The *Magic Internet Kit Programmer's Guide* assumes that you are familiar with the basics of TCP/IP and related protocols. If you need an introduction, we recommend you read *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 using the modem or serial port.

This document also assumes that you are somewhat familiar with Magic Cap development. If you need to find out more about the basics of writing a Magic Cap application, see Barry Boone's *Magic Cap Programmer's Cookbook*. The *Cookbook* steps you through the Magic Cap concepts that you need to know. Additionally, you can locate more information from General Magic's *Magic Cap Development Overview* web page located at:

`http://www.genmagic.com/Develop/MagicCap/Overview/index.html`

One specific aspect of writing a Magic Cap application, exception handling, is also very important for using the Kit. For more information on how to use exceptions, read the “Handling Exceptions” chapter of *Magic Cap Concepts*. Don’t worry if you are new to exceptions; the code snippets and templates provided with the Magic Internet Kit illustrate how to write the handlers you need. The online version of *Magic Cap Concepts* is located at:

`http://www.genmagic.com/Develop/MagicCap/Docs/Concepts/index.html`

About this document

The *Magic Internet Kit Programmer’s Guide* documents the Kit from the high-level APIs to the low-level device drivers. The *Guide* starts with a discussion of the automated tools and templates available for creating a package, and then proceeds on to discussing the high-level APIs for communications work. Later chapters dive into detail about TCP/IP and multithreaded communications functionality. This document also includes a reference section for all of the communications classes that are included in the Magic Internet Kit.

Note: This document is constantly evolving, but our web site will always have the latest version. As of this writing, the Reference section is not yet completed, and the “In Depth” chapters still have room to grow. Be sure to check our web site for the latest copy of the *Guide* at:

http://www.genmagic.com/Develop/MagicCap/Internet/index.html

Creating a New Package

Automated package creation

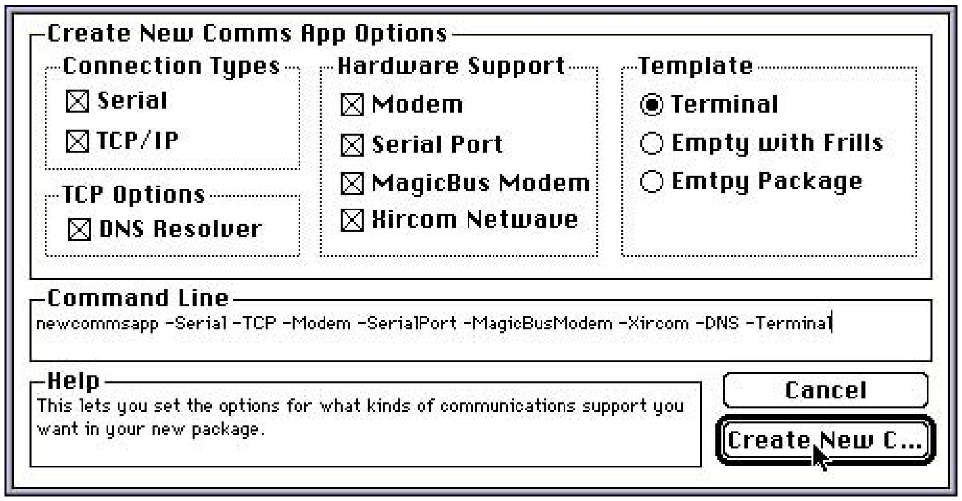
The typical first step in creating a Magic Cap application is cloning an existing package. The Magic Internet Kit takes this much farther, though, and allows you to customize your new communicating application's features with an easy-to-use dialog box.

Additionally, MIK provides several template packages to start from, so you can pick the perfect starting point for your own development. These templates start at the most basic "empty" communications package and go up to a dumb terminal package with complete user interface.

Note: If you are already familiar with developing Magic Cap applications, you are probably familiar with the **Clone Package** feature of the Magic Developer environment. You do not want to use this script when cloning a MIK template; use the **New Comms App** feature instead. The template make files that come with the Magic Internet Kit are designed to be modified by the new menu item's script, so it's best not to mess with them directly.

The New Comms App dialog

MIK's *New Comms App* dialog is accessed from the **Magic▸NewCommsApp** menu in your Macintosh Programmer's Workshop (MPW) environment. From here you can click on the checkboxes for the types of features you want in your new application.


Create New Comms App Options

Connection Types: [✓ Serial] [✓ TCP/IP]

TCP Options: [✓ DNS Resolver]

Hardware Support: [✓ Modem] [✓ Serial Port] [✓ MagicBus Modem] [✓ Xircom Netwave]

Template: [✓ Terminal] Empty with Frills Emtpy Package

Command Line

```
newcommsapp -Serial -TCP -Modem -SerialPort -MagicBusModem -Xircom -DNS -Terminal
```

Help

This lets you set the options for what kinds of communications support you want in your new package.

Figure 2 NewCommsApp script dialog

Note: If you are familiar with MPW, you'll notice that this dialog box is really a **Commando** interface to a MPW script.

Connection types and hardware support

The first part of *New Comms App* to look at is the **Connection Types** section. There are two options in here: Serial and TCP/IP. **Serial** means direct serial communications; for example using the serial port to read and write raw bytes of data. The **TCP/IP** connection type uses Transmission Control Protocol/Internet Protocol to communicate with remote hosts, for example a web server.

The next section to look at, **Hardware Support**, deals with what physical means that you'll use for communicating. These options are dependent on what connection types you have set up, for example you cannot use the Xircom Netwave PC Card for a direct serial connection. Don't worry if you're not familiar with all of the hardware or connection types; these will all be discussed shortly. The following tables illustrate how the hardware and connection types are interrelated:

Hardware	Serial Connection
Built-in Modem	Serial modem
Serial port	Direct serial via MagicBus port
MagicBus Modem	Serial via external modem on MagicBus port
Xircom Netwave	(not available)

Figure 3 Serial connection types vs. hardware support

Hardware	TCP / IP Connection
Built-in Modem	PPP via built-in modem
Serial port	(not available)
MagicBus Modem	PPP via external modem on MagicBus port
Xircom Netwave	Wireless Ethernet via Xircom Netwave card

Figure 4 TCP/IP connection types vs. hardware support

In the above table, the Built-in modem refers to the internal modem that is present in all Magic Cap communicators. The serial port refers to using a communicator's MagicBus port as a passive serial port. The next two options, MagicBus modem and Xircom Netwave, use custom drivers that are included with the Magic Internet Kit. These drivers are provided as source code so that you can learn from and modify them if you so desire. The MagicBus modem driver lets you use an external modem connected to the MagicBus port via the same APIs as the internal modem. For example, you can use this driver to provide native support for Metricom's Ricochet wireless modem. The Xircom Netwave driver allows you to use these PC Cards as a data link server for MIK's TCP/IP stack. These two drivers will be discussed in much greater detail in later chapters.

The third part of *New Comms App* to consider is the section for **TCP Options**. This section only contains one checkbox for the DNS resolver. As you might imagine, this option only applies to TCP/IP connections; if you don't use TCP/IP, this box will be grayed out. If you are using TCP/IP, DNS will allow you to specify symbolic host names as the destination for a TCP stream instead of needing IP numbers. For example, "www.genmagic.com" could be used as a valid destination name instead of it's IP address, which is currently 192.216.22.142.

The final section of *New Comms App* is used to select the template to use when creating the new package. The following section explains the features of each template.

Magic Internet Kit templates

Now that you know how you're going to communicate, it's handy to start with some work already done for you, and that's exactly what MIK's application templates are for. Say you want to brew up an application, and you want to let the user set up various options for the connection. You could then find a template with an interface that's close to what you want, and use this as the starting point for your new application.

This section will discuss the templates that are included with the Magic Internet Kit so you can decided which ones you want to use for your own projects.

Terminal (a.k.a. CujoTerm)

This is the classic TTY terminal package. When you use CujoTerm as your template, you get a simple terminal that you can connect to a remote host over whatever connections means you set up in the *New Comms App* dialog. This is a good template to use for testing out code or investigating protocols. For example, if you wanted to write a package that knew how to send email via SMTP, 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.

CujoTerm comes with a user interface for setting up connections, and you may find these components useful in your own application. The UI for setting up the connection is totally independent of the terminal code, so you can tear up the other parts of CujoTerm and the setup scene will still work.

Another key component of CujoTerm is its architecture; this terminal, while basic in user-level features, has an advanced multi-threaded structure behind it that can even support multiple simultaneous TCP/IP connections. The two key classes that implement this functionality are the **CommsActor** and **CommsManager**. The code that executes on its own thread is part of the CommsActor, and the CommsManager is used to easily create and destroy CommsActors. Both of these classes will be discussed in much greater detail in the Multithreaded Communications chapter of this document.

EmptyWithFrills

The EmptyWithFrills package is similar to the **Empty** template discussed below, but it also contains some user interface and multithreading frills from CujoTerm. The UI in EmptyWithFrills is the same as CujoTerm’s setup scene. This allows you to easily create or prototype applications without worrying about setting up the connection options. Additionally, EmptyWithFrills includes the CommsActor and CommsManager classes from CujoTerm, but the terminal-specific code is removed so that you can easily add in your own functionality.

Empty

As you might imagine, the Empty template is pretty sparse. This package is the same as Magic Developer’s EmptyPackage template, except that this version has its MPW make file already set up to communicate using the means that were selected in the *New Comms App* dialog box.

Where to go from here

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

Connecting to the World

The heart of the Magic Internet Kit is the 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 interfaces that your packages can use to create, use, and destroy links to remote hosts.

The ConnectableMeans class

In Magic Cap, a **Means** subclass specifies details about a connection that your application wants to create. For example, a **TelephoneMeans** object holds a phone number for a basic serial modem connection. These objects don't actually do the connecting, rather they just specify parameters.

The Magic Internet Kit takes this concept of a Means object much farther by defining a **ConnectableMeans** class. This is a Means object mixin that knows how to connect itself to the outside world in some way. Due to the fact that this class is a mixin, it is never instantiated directly, but rather is "mixed in" with existing means objects. The resulting subclasses then implement the methods defined by ConnectableMeans.

All of the Magic Internet Kit's connection classes inherit from ConnectableMeans. For example, the **DialupPPPMeans** object implements a version of ConnectableMeans that knows how to create TCP/IP connections using PPP over a Magic Cap communicator's built-in modem.

Source Code Note: The ConnectableMeans class definition can be found in the Means:ConnectableMeans.Def file.

Methods of ConnectableMeans

There are three methods defined by ConnectableMeans: CanCreateConnection, CreateConnection, and DestroyConnection. The methods do what they sound like, and we'll cover what each one does in detail.

CanCreateConnection

```
operation CanCreateConnection(): Boolean, noMethod;
```

This method is called by client code to determine if a connection could potentially be established. For example, if this ConnectableMeans subclass communicates using the modem, CreateConnection could check to see if the phone line is plugged in, make sure the user has set up a dialing location, and also check that the modem is not already in use.

The application calling CanCreateConnection should be aware that the boolean return value is not a guarantee that a connection attempt will succeed. There are many factors that can effect the success of a connection attempt, and many of these cannot be judged until the attempt is actually made. For example, if the object communicates using the modem, and the remote server is not answering, the object won’t know that until after dialing. CanCreateConnection therefore should only be counted on as a “sanity check” before doing any real work to try connecting.

CreateConnection

```
operation CreateConnection(): Object, noMethod;
```

CreateConnection is the method used to connect a ConnectableMeans subclass to a remote host. If the connection attempt succeeds, this method will return the Object ID of the new stream to be used. The stream returned from CreateConnection 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, CreateConnection will throw a **ConnectException** heavyweight exception. Heavyweight exceptions are somewhat different than typical lightweight exceptions used in other parts of Magic Cap, so these will be discussed shortly in the following CreateConnection error handling section.

Note: With TCP/IP connections, you can actually call CreateConnection repeatedly to get multiple streams. See the chapter on TCP/IP Communications in Depth for more information.

DestroyConnection

```
operation DestroyConnection(stream: Object), noMethod;
```

DestroyConnection is the inverse of CreateConnection; whenever a stream is no longer needed, this method is used to destroy it in whatever manner is appropriate for the specific ConnectableMeans subclass. This method should also destroy any objects or buffers that were allocated by CreateConnection. For example, if CreateConnection creates a TCPStream object, DestroyConnection will be sure to destroy that object.

CreateConnection error handling

As a general rule with communications, connecting to a remote host requires quite a few things to all work together with each other, and therefore there are many potential places for things to go wrong. Errors in Magic Cap are typically handled with **lightweight exceptions** that get thrown when the error occurs and are then caught by code designed to specifically handle that error. The lightweight exception itself is really just an unsigned value that translates into an error code listed in the Exceptions.h file in the Magic Cap interfaces. For more information on handling these types of exceptions, see *Magic Cap Concepts*.

With the multitude of possible errors that could happen when connecting, an application using the lightweight exception model would have to catch almost a dozen distinct types of exceptions. The code for just 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 very much like a lightweight 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 CreateConnection, the object that is thrown is a **ConnectException** or one of its subclasses. Code that is calling CreateConnection, therefore, needs to catch objects of the ConnectException class. The magical method that does this is called, quite reasonably, **CatchByClass**. CatchByClass will catch all exception objects of a specified class and all of its subclasses. To catch all connect-time errors, catch the ConnectException class. For only errors that deal with hardware, you can catch the HardwareConnectException class, which is a subclass of ConnectException. TCP/IP errors are thrown with TCPConnectException and IPConnectException objects, respectively. Most applications, though, will just want to catch ConnectException since it specifies all of the necessary details about the error in an attribute.

The ConnectException object has an attribute named **ConnectError**. The value of this attribute is an error code that specifies what exactly went wrong during the connection attempt. The error codes are listed in the MIKErrors.h file in the Magic Internet Kit's "Includes" file. Applications can then check for specific errors, or even check for a category of error by masking off bits. The errors are grouped by hex digit, so to see if the error had something to do with DNS, for example, a simple bitwise AND like the following will do the trick:

```
if (error & errorDNSMask) /* it's a DNS error */
```

Now let's take all of this information and see what it looks like in real, live code.

Using ConnectableMeans: an example

A quick example of using a ConnectableMeans subclass is in order. In this example, we will connect the Magic Cap communicator's built-in modem to a remote host. This will be a simple serial connection, so the only information we need about the remote host is its phone number.

The first step in this example is creating the objects that we'll need. The ModemMeans object is perfect for what we need to do; it's designed exactly for doing direct serial connections using the modem. For more information on this class, see the Serial Communications in Depth chapter of this document.

The following code defines and creates the ModemMeans object and the Telenumber object that holds the text of the destination number.

```
ObjectID means      = NewTransient(ModemMeans_, nil);
ObjectID telenumber = NewTransient(Telenumber_, nil);
ObjectID stream;
ObjectID exception;
```

Now we can “fill in” the Telenumber and set the means object’s Telenumber attribute appropriately.

```
ReplaceTextWithLiteral(Telephone(telenumber), "(800) 555-1212");
SetTelenumber(means, telenumber);
```

Now we get to the fun part. Let’s call CanCreateConnection to see if we have a chance at connecting.

```
if (!CanCreateConnection(means))
{
    /* error case */
    return false;
}
```

Once that test is passed, we’ll want to set up error handling code for CreateConnection. To do this, we’ll use the CatchByClass method mentioned above.

```
if ((exception = CatchByClass(ConnectException_)) != nilObject)
{
    /* The error code is specified by the ConnectError attribute. */
    Unsigned error = ConnectError(exception);

    /* Figure out the error type */
    switch (error)
    {
        /* Do what's appropriate here for the errors listed in */
        /* MIKErrors.h */
    }

    /* The exception object should be destroyed now that we're done */
    /* with it. */
    Destroy(exception);

    return false;
}
```

If we’ve gotten this far, call CreateConnection to connect the modem to the remote host.

```
stream = CreateConnection(means);
```

At this point we have a real, live connection. In the error case, CreateConnection failed up to the above exception handler, so our code can go on assured that the connection attempt worked. Let’s commit the exception handler and then write some bytes to the stream.

```
Commit();

Write(stream, "hello there!", 12);
```

Finally, let’s destroy the connection. For this example, we won’t need the means object anymore, so destroy that, too.

```
DestroyConnection(means, stream);
Destroy(means);
```

MIK’s ConnectableMeans subclasses

As mentioned above, the Magic Internet Kit comes with several subclasses of ConnectableMeans for you to use. These can be broken down into two categories: serial based and TCP/IP based. Each category and its classes will be discussed individually in the following sections.

Serial communications

Serial communications, for MIK’s purposes, will be defined as opening a raw data stream to a remote host without the aid of software protocols like TCP/IP. This includes slamming bits down to the serial port or connecting the modem to a remote host. There are three ConnectableMeans subclasses that are serial-based: ModemMeans, MagicBusModemMeans, and SerialPortMeans. Each of these classes will be discussed briefly, and for more information see the Reference section of this guide.

ModemMeans

Class ModemMeans, as mentioned earlier in the ConnectableMeans example code, communicates using a Magic Cap communicator’s built-in modem. If a connection attempt is successful, the stream returned from CreateConnection will be the iModem indexical. Application code can read and write from iModem since it inherits from **CommunicationStream**, and then DestroyConnection knows how to properly disconnect it.

ModemMeans inherits from **TelephoneMeans** and mixes in **ConnectableMeans**. This class does not add any additional fields, so a typical instance of this class looks the same as a TelephoneMeans instance, for example:

```
Instance ModemMeans 402;
 reservationTime: 0;
 telephoneNumber: (Telenumber 403);
End Instance;

Instance Telenumber 403;
        country: 1;
      extension: nilObject;
      telephone: (Text 404);
End Instance;

Instance Text 404;
    text: '(800) 555-1212';
End Instance;
```

An example use of ModemMeans would be to talk to a bulletin board system (BBS) that allows direct dial-in via modem. Then the client application could do whatever it needed to, e.g. process mail or files, and disconnect. One advantage of this type of connection is that it is simple to set up and implement. On the flip side of this, though, is that it is not as functional as a modem connection using Point-to-Point Protocol (PPP), especially since a raw modem stream does not have inherent error correction and recovery built in, whereas PPP does.

Source Code Note: The ModemMeans class is defined in Means:ModemMeans.Def and implemented in Means:ModemMeans.c

---

MagicBusModemMeans

This class is much like the ModemMeans class above, except that it uses the Magic Internet Kit's custom external modem driver in place of the communicator's built-in modem. The **MagicBusModem** driver allows users to plug an external modem into their communicator's MagicBus port and then the application can use it with this class.

Source Code Note: The MagicBusModemMeans class is defined and implemented in Means:MagicBusModemMeans.(Def, c)

SerialPortMeans

Class SerialPortMeans allows applications to communicate via a Magic Cap communicator's MagicBus port. The MagicBus port, when used for passive serial connections, can communicate at speeds up to 38.4 Kbps. This class inherits from **Means** and **ConnectableMeans**, and it defines its own attribute BaudRate for the speed of the connection. A typical instance of SerialPortMeans looks like the following:

```
Instance SerialPortMeans 452;
   baudRate: 38400;
End Instance;
```

Source Code Note: The SerialPortMeans class is defined and implemented in Means:SerialPortMeans.(Def, c)

TCP/IP communications

TCP/IP is the standard protocol of the Internet. Internet Protocol (IP) is a packet-based protocol that is implemented over various types of data links, for example Ethernet, and is the low-level heart of the Internet. Transmission Control Protocol (TCP) runs on top of IP and provides applications with a reliable stream interface for communications. Many applications use stream-based services, such as the World Wide Web, by means of TCP/IP. The Magic Internet Kit also supports User Datagram Protocol (UDP), and this will be discussed in the TCP/IP Communications in Depth chapter of this document.

The Magic Internet Kit provides three classes for communicating via TCP: DialupPPPMeans, MagicBusPPPMeans, and XircomMeans. Each of these classes will be discussed briefly, and for more information see the Reference section of this guide.

DialupPPPMeans

Class DialupPPPMeans is used to create TCP/IP connections using Point-to-Point Protocol (PPP) over the Magic Cap communicator's built-in modem. This is the most common way to connect a communicator to the Internet, and for good reason: it provides a reliable, error-correcting, and authenticating stream over built-in hardware. Additionally, most all Internet Service Providers support PPP, so any application using it will have excellent connectivity with existing dial-up servers.

DialupPPPMeans inherits from **DialupIPMeans**, **ConnectableMeans**, and **UsesTCP**. DialupIPMeans is a built-in Magic Cap class that contains fields for useful information, namely the remote phone number, host name, and TCP port. ConnectableMeans, as described above, adds on the single-API interface for creating a destroying connections. The UsesTCP class is provided in MIK to do the hardware independent work of managing data link servers and TCP streams. UsesTCP will be described later in the Behind the scenes with TCP/IP section.

Instances of TCP-based Means objects contain a bit more information than the serial-based Means objects that were shown above. You'll notice that some parts are the same as for ModemMeans, namely the Telenumber and Text object. There are a few additional classes, though: a FixedList and Monitor. These will be discussed in greater detail as part of the UsesTCP mixin class. Here's a sample instance of DialupPPPMeans and its supporting classes:

```
Instance DialupPPPMeans 502;
  reservationTime: 0;
  telephoneNumber: (Telenumber 503);
         hostName: (Text 505);
             port: 7;
       linkAccess: (Monitor 508);
        linkClass: PPPLinkServer_;
    meansSourceIP: 0;
       dNSServers: (FixedList 509);
            login: (Text 506);
         password: (Text 507);
End Instance;

Instance Telenumber 503;
        country: 1;
      extension: nilObject;
      telephone: (Text 504);
End Instance;

Instance Text 504;
       text: '(800) 555-1212';
End Instance;

Instance Text 505;
      text: 'www.genmagic.com';
End Instance;

Instance Text 506;
      text: 'none';
End Instance;
```

```
Instance Text 507;
     text: 'none';
End Instance;

Instance Monitor 508;
      next: nilObject;
     count: 1;
      user: nilObject;
End Instance;

/* List of DNS servers */
Instance FixedList 509;
     length: 1;
      entry: 0x12345678;
End Instance;
```

An example use of DialupPPPMeans, besides the obvious one of connecting to the Web or other Internet services, would be to take advantage of its error correction and multiple stream facilities for a vertical market application. Say that a salesperson is in the field with a Magic Cap communicator, and they need to synchronize records with the main office. A TCP/IP solution with PPP would be perfect because of the reliable connection that these protocols offer. Additionally, since TCP/IP/PPP is a totally cross-platform standard, one could write client applications for any other platform and still talk to the same server.

Source Code Note: The DialupPPPMeans class is defined and implemented in Means:DialupPPPMeans.(Def, c)

MagicBusPPPMeans

Like the serial-based MagicBusModemMeans above, MagicBusPPPMeans is just an extension of a modem-based class that uses an external modem instead of a communicator's built-in modem. MagicBusPPPMeans inherits from DialupPPPMeans and mostly makes sure that modem-specific method calls are pointed at MIK's iMagicBusModem indexical instead of Magic Cap's iModem. This is accomplished by overriding the **Stream** attribute that is inherited from the Means superclass. The Stream attribute specifies the hardware driver to use for communications.

Source Code Note: The MagicBusPPPMeans class is defined and implemented in Means:MagicBusPPPMeans.(Def, c)

XircomMeans

This class is designed for use with MIK's special driver for Xircom Netwave wireless ethernet cards. This driver is not yet heavily tested, so developers using it are currently warned to take note that they are doing so at their own risk. This driver will hopefully be polished and production-quality in the near future.

WARNING! The XircomMeans class is currently unsupported. Also, do not use Xircom Netwave PC Cards in Sony Magic Link PIC-1000 devices; the card's power requirements greatly exceeds what the PIC-1000 will support.

XircomMeans, like DialupPPPMeans, inherits from DialupIPMeans, ConnectableMeans, and UsesTCP. The DialupIPMeans inheritance is not totally appropriate since this is not a dial-up connection, but the fields for the remote host name and port are required by the TCP/IP libraries. The Telenumber attribute is ignored and should be set to nilObject.

Source Code Note: The XircomMeans class is defined and implemented in Means:XircomMeans.(Def, c)

Behind the scenes with TCP/IP

In order to minimize code duplication, the above hardware-specific TCP/IP Means classes only do "real work" that is specific to the hardware they support. If that's the case, then who's doing all the rest of the work? This section will describe the key class that lies between the high-level Means classes and the low-level TCP/IP stack: **UsesTCP**.

UsesTCP is the mixin class that all of the above TCP/IP Means classes inherit from to do the gnarly work of managing TCP/IP streams and data link servers. UsesTCP knows how to automagically connect and disconnect the link servers when needed, resolve host names with DNS when they are specified symbolically, and other fun stuff. We'll take a look at the features of UsesTCP, and you can find complete documentation on its implementation in the TCP/IP Communications in Depth chapter.

First of all, TCP/IP has the neat capability of creating multiple communications streams over the same data link. This would usually present a bit of a problem, though, when it comes to connecting and disconnecting streams because the code would have to figure out when it needs to create or destroy the link server. UsesTCP does this automatically by keeping count of the number of streams that a given link server has running on it. Client code, therefore, does not have to worry about managing the data link.

Second, the names of IP destinations can be specified by either an IP address, e.g. 192.216.22.142, or symbolic host name, e.g. www.genmagic.com. UsesTCP will look at the **HostName** attribute of the Means object, and if the name is not an IP address, it will ask the DNS servers in the **DNSServers** attribute if they can turn the name into an IP address.

A third feature of UsesTCP is its flexibility. UsesTCP provides methods that subclasses can override to do custom initialization of the data link server and do any negotiation with the remote host before the link level protocol is activated. Additionally, client code can manually open and close the data link level without creating a TCP/IP stream at the same time. This is useful if the client code wants to create and destroy several streams, one after another, without opening and closing the data link each time.

For more information on UsesTCP, see the TCP/IP Communications in Depth chapter and the Reference section of this document.

Source Code Note: The UsesTCP class is defined and implemented in Means:UsesTCP.(Def, c)

Example: *Finger* Client

Enough talk, let's rock! In this section we will build a TCP/IP application from the ground up using the Magic Internet Kit. The application will be a **finger** client that asks a remote host about the status of a user on that host. For more information on the finger protocol, see RFC 742. We'll build this application in a step-by-step fashion, so boot your computer and follow along!

Step 1: Create the code base

The data that we will exchange with the server is very basic: we send a user name and the server sends back information on that user and closes the stream. For our client, there will be one text field for the user name, and another field for the server's reply. We'll also need to provide a basic user interface for setting up the information on our host. Given these requirements, the EmptyWithFrills template sounds like a great starting place, so let's open up *New Comms App* and create our new application base.

Select TCP/IP as your connection type, and then add whatever hardware support you desire. Also include DNS if you want name resolution capabilities. I'll name my new application "Finger." Now build the new application for the Macintosh simulator.

Step 2: Build the user interface

The EmptyWithFrills template provided us with a new door in Magic Cap's hallway. Behind that door is a scene with one button in it that takes the user to its connection setup scene. Now add the following components to this scene:

- TextField object for the user name
- TextField object for the server's reply
- Button object to do the work

For my version of Finger, the user interface looks like this:

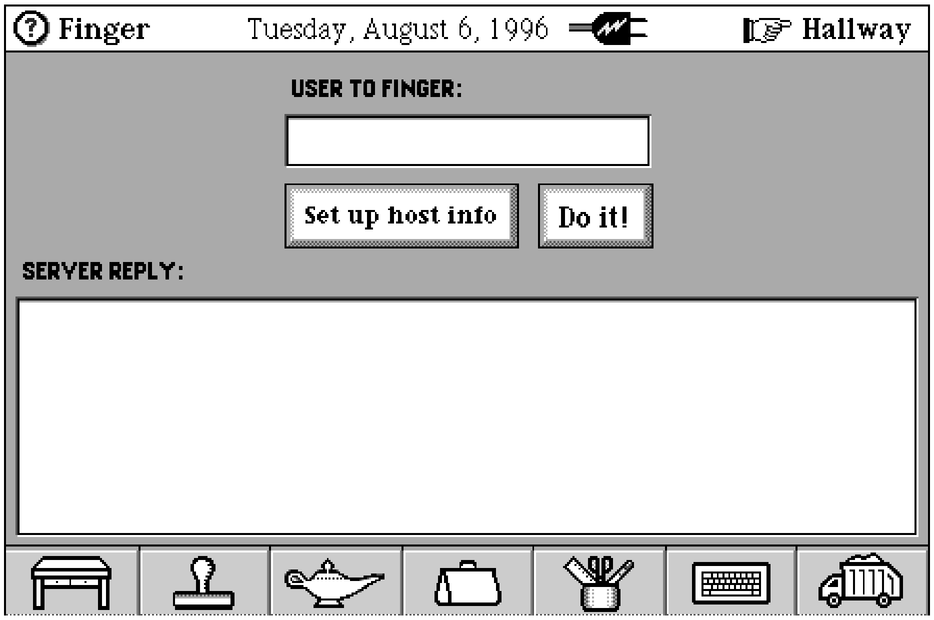


Figure 5 Finger user interface

When you have the interface the way that you like it, use Magic Cap's Inspector tool to select Finger's Scene. Then select **Examine▶Dump Inspector Target Deep** from the simulator's menu to dump the interface components in this scene. If you hold down the shift key while doing this last step, the objects will be dumped to the Macintosh clipboard, too, which will make the next step easier.

Note: We are not using Magic Cap's "Dump Package" feature since Dump Package would dump a lot more stuff than we need. For example, all of the setup components would get dumped, and these are most easily kept in separate files like the template starts with. Dump Package won't handle the multiple object instance files properly.

With these objects dumped out, go back to MPW and paste them into your Objects.Def file as you would with any other Magic Cap application.

Step 3: Define the FingerButton class

Let’s make the “Do it” button that we added above do the real work. First, subclass Button and override its **Action** method. A few fields for the two TextField objects would be handy, too, so add those. Here’s the class definition that I’ll use for this new class, called FingerButton:

```
Define Class FingerButton;
        inherits from Button;

        field           queryText: Object, getter, setter;
        field           replyText: Object, getter, setter;

        attribute       QueryText: Object;
        attribute       ReplyText: Object;

        overrides       Action;
End Class;
```

With the new FingerButton class defined, modify your Objects.Def file to reflect the new class. Here’s an excerpt from my Objects.Def file:

```
Instance FingerButton 'Do it!' 9;
           next: (TextField 'Server reply:' 10);
       previous: (SimpleActionButton 'Set up host info' 17);
      superview: (Scene 'Finger' 6);
        subview: nilObject;
 relativeOrigin: <66.5,-43.0>;
    contentSize: <50.0,24.0>;
      viewFlags: 0x70101200;
     labelStyle: iBook12Bold;
          color: 0xFF000000;
       altColor: 0xFF000000;
         shadow: nilObject;
          sound: iTouchSound;
          image: nilObject;
         border: iSquareButtonBorderUp;
      queryText: (TextField 'User to finger:' 2);
      replyText: (TextField 'Server reply:' 10);
End Instance;
```

If you like, you can also remove the “Port” setup TextFields from the object instance files related to the Means classes, e.g. DialupPPPObjects.Def. The files are, by default, in a folder called “MIKObjects” within your project’s folder. We won’t need for the user to set up the remote port number since we know that port 79 is the finger port on Unix machines.

Step 4: Write the code!

Now it's time to write the code for FingerButton's Action method. Here's the code as one block. We'll discuss it in a step-by-step fashion afterwards.

```
Method void
FingerButton_Action(ObjectID self)
{
    ObjectID    means = DirectID(iiMeans);
    ObjectID    replyText = ReplyText(self);
    ObjectID    queryText = QueryText(self);
    ObjectID    stream;
    ObjectID    exception;
    Unsigned    count;
    Boolean     done;
    Str255      replyString;
    Str255      queryString;

    /* Clear the reply TextField */
    DeleteText(replyText);

    /*
    ** Make sure the Means object is TCP-capable! iiMeans refers to
    ** the current Means object that the user set up in the "Finger
    ** Setup" scene included with the template that we used
    ** (EmptyWithFrills). See ConnectChoiceBox and HardwareChoiceBox's
    ** SetLevel methods for the implementation of this.
    */
    Assert(Implements(means, UsesTCP_));

    /* Set the port to 79 (the finger port) */
    SetPort(means, 79);

    /* If we fail the first sanity-check, return immediately. */
    if (!CanCreateConnection(means))
    {
        Honk();
        ReplaceTextWithLiteral(replyText, "Ack! Can't connect!");
        return;
    }

    /* Catch all connect-time exceptions. */
    if ((exception = CatchByClass(ConnectException_)) != nilObject)
    {
        /* An error occured while connecting! We'll just return. */
        Honk();
        ReplaceTextWithLiteral(replyText,
             "Ack! An error occured while trying to connect!");

        /* Destroy the exception object now that we're done with it. */
        Destroy(exception);

        return;
    }

    /* Connect to the remote host! */
    stream = CreateConnection(iiMeans);

    /* Commit the above exception handler */
    Commit();
    /* Send the query to the remote host */
    TextToString(queryText, queryString);

    if (Catch(serverAborted) != nilObject)
    {
        /* An error occured while writing! */
        Honk();
        ReplaceTextWithLiteral(replyText,
            "Ack! An error occured while trying to write!");
        return;
    }

    Write(stream, &queryString[1], queryString[0]);
    Write(stream, "\xD""\xA", 2);
    Commit();

    /* Read in the reply */
    done = false;
    while (!done)
    {
        char*   currentChar;

        /* Read up to 255 characters at a time */
        count = Read(stream, replyString, 255);
        replyString[count] = 0x0; /* Null-terminate the string */

        /* Turn those pesty carriage returns into spaces */
        currentChar = replyString;
        while (currentChar[0])
        {
            if (currentChar[0] == 0xd)
                currentChar[0] = 0x20;
            currentChar++;
        }

        /* Stick the data into the "server reply" TextField */
        AppendLiteral(replyText, replyString);

        if (count < 255)
            done = true;
    }

    /* Destroy the connection now that we're done. */
    DestroyConnection(means, stream);
}
```

Step 4.1: Getting ready to take-off

```
    Assert(Implements(means, UsesTCP));
    SetPort(means, 79);

    if (!CanCreateConnection(means))
        return;
```

The first step is a bit of pre-flight checking, namely checking that the Means object that the user selected in the “Finger Setup” scene implements UsesTCP. Of course, when we created the application, we only specified TCP means objects like DialupPPPMeans as ones we wanted to use, so this should never happen. We’ll also manually set the remote port to 79, which is the finger port as specified in RFC 742.

Once that’s done, we’ll call our sanity-check method CanCreateConnection to see if we can possibly connect. If not, we’ll return immediately.

Step 4.2: Set up the error handling code

```
if ((exception = CatchByClass(ConnectException_)) != nilObject)
{
    Honk();
    Destroy(exception);
    return;
}
```

Before we try to do the real work of establishing a connection, we must set up the exception handler. We'll catch all ConnectException objects and just return from our method. A real application would want to check the ConnectError attribute of the exception object to see exactly what kind of exception was thrown.

Step 4.3: Take off!

```
stream = CreateConnection(means);
Commit();
```

The CreateConnection method should do just that: create a live connection to the remote host. If this method fails, it will throw a ConnectException and wind up in our above exception handler. If not, the code will just go on. At this point we'll want to commit the connect-time exception handler since the connection worked.

Step 4.4: Send the query

```
if (Catch(serverAborted) != nilObject)
{
    /* An error occured while writing! */
    Honk();
    ReplaceTextWithLiteral(replyText,
        "Ack! An error occured while trying to write!");
    return;
}

Write(stream, &queryString[1], queryString[0]);
Write(stream, "\xD""\xA", 2);

Commit();
```

Now it's time to send our query to the remote host. Before we actually call TCPStream's **Write** method, though, we should set up error handling code. In this case, we want to catch **serverAborted** exceptions. If the Write method fails, for example if the remote server closes the stream before all the bytes are written, then Write will throw a serverAborted exception. If we don't tell Magic Cap that we can handle this type of error, the uncaught exception will cause the communicator to reset. That would be bad.

With our error handling code in place, we can try to stuff some bytes down the TCP stream. The first Write call sends the user name to the server, and the second Write sends a carriage return/linefeed combo that terminates the query. After this is done, we can **Commit** the above exception handler since we are done with the code that could fail.

Step 4.5: Read the response

```
count = Read(stream, replyString, 255);
(...)
AppendLiteral(replyText, replyString);
```

At this point, the server should be spitting data back at us. We'll read this in 255 byte chunks and stick it in the "Server Reply" text field. The TCPStream_Read method will block until the request can be satisfied, unless the stream gets closed under it, so we can use that to our advantage here. In the case of finger, the server will close the stream immediately after sending its data, so at that point our Read call will return with however many bytes it could get.

Step 4.6: Close the stream

```
DestroyConnection(means, stream);
```

Now that we're done with the stream, we can get rid of it. The underlying TCP code will have already disconnected the TCP stream for us since the remote host disconnected earlier after sending its data, but we still need to let the ConnectableMeans object know that we're done. In this case, the means object will disconnect the data link server that the TCP stream was using, and also destroy the TCPStream object.

Step 5: Rebuild and play with your new application

Now rebuild your copy of Finger and see if it works! If it doesn't, you can double-check it with the copy of Finger that accompanies the Magic Internet Kit in the documentation folder.

Step 6: Complain about Finger stopping user interaction

Finger still leaves a bit to be desired. For example, you'll notice that Magic Cap will stop the user from doing anything while the FingerButton_Action method is running. This is because the code is all executing on the **User Actor**, which is the thread that all user interaction runs on. A better solution would have this code executing on its own thread so that the user can go do other stuff while Finger runs. That's what we'll do next.

Multithreading with Actors

Magic Cap is a multi-threading platform, and threads are very handy for communications. As you've seen above with Finger, 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 that handles 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 *Magic Cap Concepts* provides a far more complete discussion of this topic. We recommend that you read that chapter as soon as you have time.

Actor concepts

Like everything in Magic Cap, an actor is an object. The base Actor class does not really 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 be sure to give up time to other actors. This is done by calling **RunNext** on the scheduler, referenced by the **iScheduler** indexical. Calling RunNext tells the scheduler to run the next waiting actor.

Some methods that 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 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 a bit larger than the default:

```
NewActorParameters newActorParams;

ZeroMem(&newActorParams, sizeof(NewActorParameters));
SetUpParameters(&newActorParams.header, newActorObjects);

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

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

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(ObjectID self, Parameters* UNUSED(params))
{
    while (FeelingPeachy(self))
    {
        DoSomeStuff(self);

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

There is one essential caveat to using actors that you should be aware of: code running on an actor that isn't the user actor cannot call methods that deal with drawing on the screen. This means that code in the above MyActor_Main method cannot move viewables on the screen, call RedrawNow, or otherwise change stuff on screen. If you need to mess with viewables, do so using **RunSoon**. RunSoon will be discussed shortly in the Moving between actors section.

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 when needed

There is one more interesting caveat to using actors in Magic Cap: an actor is a transient object, so in Magic Cap 1.0 a power cycle would destroy the actor. This is not always that case in Magic Cap 1.5, so special care must be taken to ensure that the state of an actor does not get confused if the power is turned off and then back on. This is particularly important with communications, of course, because any data links will get shut down when the power is turned off.

The key to managing transient actors in Magic Cap 1.5 is overriding the **Reset** method of an object. Reset 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.

Source Code Note: Two of MIK's template packages illustrate how to override reset and destroy remaining actors at power-up time. See the CommsManager class in the Terminal and EmptyWithFrills templates.

Destroying an actor

Code can destroy an actor using Magic Cap's **Destroy** 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.

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. For more information on these topics, see the *Magic Cap Concepts* chapter on actors.

Semaphores

A **Semaphore** object is very handy for controlling access to a resource. Magic Cap's semaphores are quite a bit more intelligent than the typical 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 in the Semaphore class that you should be aware of: **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 accessed, though, Access will block until someone else releases the semaphore using Release. Release will tell the semaphore that you're done messing with it, and it will then wake up the next actor in the queue, if any, that wanted 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 *Magic Cap Concepts* 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 a serverAborted exception thrown on a communicating application's comms actor won't be caught accidentally by the Post Office actor in the system.

If code executing on one actor wants to throw an exception on a different actor, 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.

RunSoon

The RunSoon method lets code in one actor specify a completion function that is to be run on the User Actor. When Magic Cap's scheduler switches to the User Actor, this function will be executed. For example, if you are creating Card objects from data that is being pulled in by a communications stream, you can only perform some of the Card class's methods from the User Actor, e.g. Borrow/ReturnForm. The best way to do this, then, would be to read the data in from one actor and then call RunSoon to perform the final card creation from the User Actor.

Usage of this method is most easily explained with an example:

```
typedef struct
{
    Parameters  header;
    ObjectID    interestingThing;
} DoSomethingUsefulParams;
#define numObjectsDoSomethingUsefulParams 1

Private void
DoSomethingUseful(DoSomethingUsefulParams *params)
{
    ObjectID thing = params->interestingThing;
    DestroyTransientBuffer(params);

    DoSomethingElse(thing);
}

Method void
Thing_DoSomethingElse(ObjectID self)
{
    // stuff...
}

#undef  CURRENTCLASS
#define CURRENTCLASS MyActor

Method void
MyActor_Main(ObjectID self, Parameters* UNUSED(params))
{
    DoSomethingUsefulParams*    params;
    // other stuff...

    params = NewTransientBuffer(sizeof(*params));
    SetUpParameters(&params->header, numObjectsDoSomethingUsefulParams);
    params->interestingThing = ThingObject(self);

    RunSoon(true, (CompletionFunction) DoSomethingUseful, &params->header);
    // other stuff...
}
```

Step 1: Define the parameters for the RunSoon completion function

```
typedef struct
{
    Parameters  header;
    ObjectID    interestingThing;
} DoSomethingUsefulParams;
#define numObjectsDoSomethingUsefulParams 1
```

RunSoon completion functions can take a parameter block structure like the one used here. The first item in this structure is a **Parameters** header. This is used for type checking, so you don't have to fill anything in for it. After the header, you can stick anything you want in the structure. Keep in mind the number of ObjectIDs that you use, since you'll need this later. A handy trick is to define a symbol for the number of ObjectIDs as shown above.

Step 2: Write the completion function

```
Private void
DoSomethingUseful(DoSomethingUsefulParams *params)
{
    ObjectID thing = params->interestingThing;
    DestroyTransientBuffer(params);
    DoSomethingElse(thing);
}
```

Completion functions for RunSoon calls are typically very short and merely call another method of the class in question for doing the “real work.” When a completion function is executed, the current context may not be the on that you expect, e.g. your own package. This is usually not what you want, but calling a method of one of your objects will ensure that Magic Cap switches into your package’s context.

Step 3: Write the methods that do the real work

```
Method void
Thing_DoSomethingElse(ObjectID self)
{
    // stuff...
}
```

As mentioned in the previous step, do the “real work” of a RunSoon in a method of one of your package classes and not the completion function.

Step 4: Call RunSoon from another actor!

```
Method void
MyActor_Main(ObjectID self, Parameters* UNUSED(params))
{
    DoSomethingUsefulParams*    params;

    params = NewTransientBuffer(sizeof(*params));
    SetUpParameters(&params->header,
        numObjectsDoSomethingUsefulParams);
    params->interestingThing = ThingObject(self);

    RunSoon(true, (CompletionFunction) DoSomethingUseful,
        &params->header);
}
```

Now that your completion function is all set up, you can now call RunSoon from any actor that you want, and the completion function will execute on the User Actor.

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 again. MIK includes actors in two of its templates, Terminal and EmptyWithFrills. In Terminal, all connecting and reading is performed on the CommsActor object. Additionally, there is a CommsManager class that is used to create and destroy CommsActor objects. The EmptyWithFrills template provides these same classes, except the terminal-specific code is removed and a “your code goes here” comment is in its place.

Source Code Note: See the CommsActor.(Def,c) and CommsManager.(Def,c) files in the CujoTerm and EmptyWithFrills templates for their uses of Actors.

Serial Communications in Depth

Serial communications, for the purposes of the Magic Internet Kit, will be defined as using a raw data stream without intervening protocols like PPP. For example, using the modem to directly dial to a bulletin board system (BBS) or using the MagicBus port as a serial port counts as serial communications.

This chapter of the MIK Programmer's Guide will discuss the lower-level APIs that are built into Magic Cap for these types of serial communications. These APIs are below the level of the Magic Internet Kit framework, but the information presented here is valuable regardless of what level of APIs your application is using.

Using ConnectableMeans vs. low-level APIs

The first issue to discuss is which level of programming interfaces that your application should use. The low-level communications code in the system is what developers formerly had to use, but the Magic Internet Kit provides a higher level of abstraction that lets the developer program with one easy-to-use API and let the MIK framework worry about the messy and means-specific details.

In general, programming to the MIK framework is preferable to writing code directly for the low-level APIs. The Magic Internet Kit was designed to be both lightweight and modular, so your application only has to use the parts that it wants. In this case, selecting only serial communications support when creating your application will ensure that only the serial support classes are included, so there will be minimal overhead even though you get the fancy MIK interfaces.

The Serial and TCP/IP In Depth chapters are included with the Magic Internet Kit documentation since they may lend insight into the design choices made in the implementation of the MIK framework. Additionally, if you're trying to debug a comms problem, the information presented here can be extremely valuable. For these reasons, you should read these chapters even if you are using the high-level MIK framework.

Note: Pay special attention to the Detecting Loss of Carrier section; this information is very useful for all applications that use the modem.

Introducing the SerialServer and Modem classes

There are two key classes that you should know about for low-level serial communications: **SerialServer** and **Modem**. Both of these classes are used for accessing a communicator's built-in modem, and the SerialServer is used by itself for accessing the MagicBus port as a serial port.

The SerialServer is the lowest-level class that is used for serial communications, and this provides essential methods like **Read**, **Write**, **OpenPort**, and **ClosePort**. There are two serial servers in Magic Cap, referenced by the iSerialAServer and iSerialBServer indexicals, for the modem and MagicBus ports, respectively. You will rarely use iSerialAServer directly, but instead use the iModem indexical to access the modem-specific APIs.

The Modem class provides more functionality than the core serial server that is specific to using a Magic Cap communicator's built-in modem. For example, Modem's **Connect** and **Disconnect** methods will do special stuff like dialing a phone number and ensuring proper configuration of the modem. Essentially, the Modem class sits on top of the SerialServer class and worries about most of the lowest level details for you.

Using the modem

Using Magic Cap's modem without the aid of the high-level MIK framework is pretty easy; there isn't that much to be done, but you still have to make sure that you do it right. We'll first discuss basic modem usage, such as connecting and disconnecting, and then discuss related topics like detecting loss of carrier.

Connecting the modem

This part looks pretty complex, but in reality connecting the modem is a one-line operation with lots of error handling code around it. Connect time errors are all handled by means of **exceptions**, so if you are not familiar with exception handling you should read the "Handling Exceptions" chapter of *Magic Cap Concepts* before proceeding.

Step 1: See if there's a chance at connecting (preflighting)

The first step in trying to connect is to make sure that there's a chance that the connection could succeed. There are a few situations to check for that might immediately cause failure, so it's best to check for them before doing anything else.

First, check to see if the user has already set up a dialing location. If this is not set up, the Modem's Connect method will surely fail since it tries to be smart about prepending area codes and/or long-distance access numbers. This is the code that you should use to check for this case:

```
if (!SetupDialingLocation(iSystem)) /* fail! */
```

Second, check to make sure that the phone line is plugged in. Modem's CanConnect method will determine if this is the case:

```
if (!CanConnect(iModem)) /* fail! */
```

Third, check to make sure that someone else is not already using the modem:

```
if (InUse(iModem)) /* fail! */
```

Note that these checks do not guarantee that the above situations could not arise between checking here and trying to connect later. That's why we still need to put full-strength error handling on the connection itself. The above checks should still be made to make sure that we can detect obvious error cases before trying an all-out connect.

Source Code Note: Examples of using these "preflight checks" can be found in each of MIK's ConnectableMeans subclasses as part of the CanCreateConnection methods.

Step 2: Set up a TransferTicket object

The Modem class's Connect method expects a TransferTicket object as its second parameter, so we need to create one. The Means attribute of this object is the only one that we need to worry about, so don't be worried about the zillions of other fields. We'll create our ticket on the fly in our code, but as an alternative you could define one in your package's object instance file.

```
ObjectID phoneNumber = NewTransient(Text_, nil);
ObjectID telenumber  = NewTransient(Telenumber_, nil);
ObjectID means       = NewTransient(TelephoneMeans_, nil);
ObjectID ticket      = NewTransient(TransferTicket_, nil);

ReplaceTextWithLiteral(phoneNumber, "(800) 555-1212");
SetTelephone(telenumber, phoneNumber);
SetTelenumber(means, telenumber);
SetMeans(ticket, means);
```

Step 3: Set up the exception handlers

There are two main exceptions that you should expect to catch if anything goes wrong with connecting the modem: **cannotOpenPort** and **commHardwareError**. The first exception will be thrown if the modem port cannot be opened for some reason. The second exception will be thrown if something bad happens during or immediately after the connection process, for example if the remote modem does not answer the call. Here’s the code that you need for catching these exceptions:

```
if (Catch(commHardwareError) != nilObject)
{
    /*
    ** If we are here, a commHardwareError exception got thrown.
    */
    return false; /* do what's appropriate here */
}
if (Catch(cannotOpenPort) != nilObject)
{
    /*
    ** If we are here, a cannotOpenPort exception got thrown. Note
    ** that the commHardwareError above did _not_ get thrown, so
    ** we need to do one Commit() to take it off the exception
    ** stack and commit its changes.
    */
    Commit();
    return false; /* do what's appropriate here */
}
```

Step 4: Try to connect!

Now that all the error handling code is in place, it’s time to try connecting. Modem’s Connect method is the method that we want to use.

```
Connect(iModem, ticket);
```

Note: Don’t use the Modem class’s ConnectToNumber or DialNumber methods. These methods are for system use only as the setup required before calling them is hardware dependent.

Step 5: Commit the exception handlers and clean up

If the connection attempt did not fail up to the exception handlers, then the connection attempt succeeded. At this point the exceptions handlers should be committed. The transfer ticket and its associated objects could also be destroyed as well unless you intend to use them again later.

```
Commit();        /* cannotOpenPort */
Commit();        /* commHardwareError */

Destroy(ticket); /* if you want to */
```

Using a live modem connection

Once the modem is connected, using it is pretty straightforward. The API is that of Magic Cap’s **Stream** mixin class; **Read** is used for reading, **Write** is used for writing, and **CountReadPending** returns the number of bytes that are available for reading. One important note is that these methods are all synchronous, i.e. they block until they are completed. This blocking is not a big deal for writing to the modem since writing is usually a fast operation that does not have to wait for the other end. For reading, though, the issue is more important. If the remote server takes its time in sending you the data that you are expecting, then you either have to poll

CountReadPending to avoid blocking the User Actor or run your code on its own thread. For more information on using your own thread, see the Multithreading with Actors section of this document.

If any of the stream methods cannot complete their task, they will return with whatever they could get done. If the method failed to complete due to a loss of carrier, for example the other end hanging up, then your code will have to detect this separately. See the later section on Detecting Loss of Carrier for more information.

Disconnecting the modem

Fortunately, disconnecting the modem is easier than connecting it. In typical usage, the Modem class's **Disconnect** method does the right thing. There is one interesting caveat here: if your code is blocked on a Read call, Disconnect will not return until the Read is completed. If you don't want to wait for the Read to return, or you know that the Read will never return, then disconnecting the modem is a tad more complex.

To disconnect the modem when other code is blocked on it, you have to abort the serial server to release the semaphores being held down by the outstanding reads or writes. Here's how to do that and then disconnect the modem:

```
ObjectID serialServer = Target(iModem);

if (Catch(serverAborted) == nilObject)
{
    Abort(serialServer);
    Commit();
}

ClosePort(serialServer);
OpenPort(serialServer);
Disconnect(iModem);
```

The first step is to abort the serial server, which you will note is found in the **Target** attribute of the modem, iModem. This will call ReleaseAndFail with a serverAborted exception on any semaphores that may be held down in the serial server, so the code should plan ahead and catch serverAborted exceptions. After that, closing and opening the serial server should get things in a reasonable state for calling Disconnect on the modem itself.

Note: The **Abort** method of the Modem class was designed to work in the case of blocked readers, but it doesn't. Unfortunately, code like the above must be used to get the modem unstuck.

Detecting Loss of Carrier

One fun aspect of communications is that anything can go wrong at any time. For example, the user can pull out the phone cord and mess up your modem connection or the remote modem could hang up the line. Your application should be prepared to deal with these situations by telling the modem that you want notification of a change in the carrier.

To tell the modem that you want to monitor the carrier, call **MonitorDCD** on the modem object with the second parameter set to the object that you want notified. The third parameter should be **true** to tell the modem to start notifying your target object of carrier changes.

Notification of carrier changes is provided via the callback **CarrierChanged**. The second parameter passed to this method by the modem, hasCarrier, tells your code if this change is a gain or loss of carrier. A typical CarrierChanged method might look like the following:

```
Method void
MyObject_CarrierChanged(ObjectID self, Boolean hasCarrier)
{
    if (!hasCarrier && MethinksIAmConnected(self))
    {
        /* I thought I was connected, but I guess I'm not anymore! */
        CleanUpAndGoHome(self);
    }
}
```

If carrier is lost in the middle of your communications session, you should still call Disconnect on the modem to get things in their proper state. Disconnecting the modem will also reset MonitorDCD so that it does not notify any objects of carrier changes, so be sure to set up MonitorDCD before every connection attempt.

Dispelling myths and stuff to avoid

The Modem class has been greatly misunderstood in the past, so this section discusses things that should not be done. Generally, one should use only the methodologies presented above for dealing with serial communications, so if in doubt about a method not mentioned in this chapter, don’t use it.

AT commands are evil

The developer should never send AT commands directly to the modem from high level code. This is for compatibility reasons; AT commands are proprietary and each modem manufacturer’s command set typically differs. Application code should never assume the type of modem that it is running on since the iModem indexical does not always refer to a communicator’s built-in modem. Other third party applications can replace iModem with their own modem drivers, and the modems on the other end of the driver may not use the Rockwell command set that the internal modem uses.

Cruel and unusual punishment with SetBitRate

Note: This section applies only to total modem geeks.

If only one thought is to cross your mind when you wake up every morning, besides "not again," it should be "SetBitRate is evil."

This leads to an important question, namely, “huh?” HardwareStream_BitRate is a greatly misunderstood attribute lurking in Magic Cap, so an explanation of what it does is in order. The comments in the Server.Def class definition file say the following about HardwareStream's BitRate attribute: “Return current i/o bit rate, lower if mismatched.” Maybe that's a tad terse. I'll say “BitRate and SetBitRate refer to the DTE/DCE speed. Note that this is not the same as the carrier speed.” That's even more terse. Let me start by discussing some of the lower level details about modem communication.

First, There are two connections going on with device modems: the modem is talking to another modem, and it's also talking to the device CPU. I'll refer to the modem to modem speed as the **carrier speed**. I'll refer to the CPU to modem speed as the **DTE/DCE** speed. In the RS232 standard, DTE refers to Data Terminal Equipment, in this case the CPU, and DCE refers to Data Communications Equipment, or the modem. If we simply called the DTE a processor and DCE a modem, then a whole bunch of modem experts would be out of jobs since that would be too easy.

On the 2400 Baud devices like Sony's PIC-1000 and Motorola's Envoy, the modem and CPU do not hardware handshake, so the carrier speed and DTE/DCE speed have to be the same to avoid loss of data. SetBitRate(2400) sets the DTE/DCE speed to 2400 bps to make sure that the speeds are matched. Of course there's an edge case that can mess everything up: if the phone line is really noisy, or the modem on the other end is really slow, you might get a connection at 1200 Baud. In that case SetBitRate(2400) would not work.

On Sony's PIC-2000, the 14.4K modem and processor have hardware handshaking that lets them tell each other to stop sending bits for a while if one is bogged down. This hardware handshaking makes life much easier for the system; just set the DTE/DCE speed as high as it will go and let the handshaking prevent overflow. The only danger is that the DTE/DCE speed must be greater than or equal to the carrier speed, otherwise the following case could arise: the modem is connected at 14400 bps and the DTE/DCE speed is 2400. The modem can then receive data faster than it can send the data to the CPU, so it buffers everything it can. When the modem's receive buffer fills up, data loss occurs.

Here's the key: SetBitRate does not have anything to do with carrier (modem to modem) speed; it only controls DTE/DCE speed. There is no easy way to force carrier speeds in Magic Cap at the time. If you SetBitRate(2400) on a PIC-2000, it will still try to connect at 14,400.

Here's the best solution to the above complexities and problems: use Modem’s Connect method. Connect knows about the hardware that it's running on, so it will handle all the device-specific details. It will handle carrier speed fallback on 2400 Baud devices, and it will make sure the DTE/DCE speed is maxed out on others. The user may not have control over the carrier speed, but that's often a good thing. Modems know how to talk to each other, so let them handle it. If the PIC-2000 tries to talk to a 9600 Baud modem, it will know to fall back and connect at 9600. The secret is to not worry about it.

TCP/IP Communications in Depth

TCP/IP communications is a very broad topic, so this document will only focus on the Magic Cap aspects of TCP/IP that are different from equivalent implementations on other platforms. This chapter assumes that you are somewhat familiar with the workings of TCP/IP, so if you need a more information, see *Internetworking with TCP/IP* by Douglas Comer or *TCP Illustrated* by Richard Stevens.

Like the previous chapter, Serial Communications in Depth, this chapter dives below the level of the Magic Internet Kit framework and talks about the guts that MIK takes care of for you. Regardless, this information is still important for understanding what's going on under the hood of the Magic Internet Kit.

Using ConnectableMeans vs. low-level APIs

As with serial communications, the first issue to discuss is which level of programming interfaces that your application should use. The low-level communications code in the system is what developers formerly had to use, but the Magic Internet Kit provides a higher level of abstraction that lets the developer program with one easy-to-use API and let the MIK framework worry about the messy and means-specific details.

In general, programming to the MIK framework is preferable to writing code directly for the low-level APIs. The Magic Internet Kit was designed to be both lightweight and modular, so your application only has to use the parts that it wants. In this case, selecting only TCP/IP communications support when creating your application will ensure that only the proper TCP support classes are included, so there will be minimal overhead even though you get the fancy MIK interfaces.

The Serial and TCP/IP In Depth chapters are included with the Magic Internet Kit documentation since they may lend insight into the design choices made in the implementation of the MIK framework. Additionally, if you're trying to debug a comms problem, the information presented here can be extremely valuable. For these reasons, you should read the chapters that apply to your application even if you are using the high-level MIK framework.

Note: Pay special attention to the Using a TCP stream section; this information is very useful for all applications that use TCP streams. Additionally, the Managing data link servers section can be useful for applications that want to manage link servers on their own.

Link servers, protocol stacks, and streams, oh my!

TCP/IP is not a simple beast, but it can be easily mastered. There are several protocols in action that are layered on top of each other, hence the name “protocol stack,” and this section will briefly describe how they are related. Magic Cap’s object-oriented systems maps these protocol layers into objects, so this section will also discuss the objects that are responsible for implementing the functionality of each layer.

The data link server

At the bottom of the stack is an object that knows how to talk to some type of hardware. This hardware can be a modem, Ethernet card, or anything else that can exchange bits. If there was a digital interface for two tin cans and a string, a link server could be written for it.

The job of the data link server is to throw bits back and forth with a remote host. It can be as smart or a stupid as it needs to be, but a good link server would implement some type of error correction. This server then talks above itself to the next higher level in the stack: IP.

Having a data link server between IP and the hardware allows IP to be generic and run over whatever server the programmer desires. This modularity keeps IP flexible. Examples of common data link servers are dial-up Point-to-Point Protocol (PPP) and Ethernet.

In the Magic Internet Kit, **PPPLinkServer** implements the core PPP protocol and is used by classes like **DialupPPPMeans** and **MagicBusPPPMeans**. Ethernet is implemented in the **EtherServer** class and is used by the **XircomEthernet** driver class and its matching **XircomMeans** class. Both of the link servers inherit from the **DataLinkServer** class, which is the abstract class that defines the APIs that all Magic Cap data link servers should implement.

Internet Protocol (IP)

Internet Protocol, as one might expect, is the core of the Internet. This protocol defines the connectionless delivery service that is the foundation upon which transport services rest, and the interface that talks to data link servers below. IP also acts as a multiplexer/demultiplexer for the transport services, e.g. TCP, that sit on top of it. IP can handle multiple transport services, so it decides which packets go to which transport services.

In MIK, the **IPSwitch** and **IPDaemon** classes implement IP. Applications will never talk to these classes directly; they are only used by the link servers and the transport protocols.

Transmission Control Protocol (TCP)

Transmission Control Protocol is a protocol that sits on top of IP and provides a mechanism for reliable, stream-based transport. IP is an unreliable service, i.e. packets can be lost or duplicated, but TCP manages that problem and requests packet resends and throws away duplicate packets. IP is also packet-based, whereas TCP provides a stream-based interface similar to using direct serial communications. For these reasons, TCP is an extremely important protocol and is the transport service of choice for many applications.

TCPStream is the Magic Internet Kit class that implements TCP. These objects are used by the application layer, and as such will be described in much greater detail shortly.

User Datagram Protocol (UDP)

User Datagram Protocol is another transport service, like TCP, that talks to IP on the bottom end and an application on the top end. UDP is packet based and unreliable by definition, but it is useful for some types of applications. This chapter will mostly deal with TCP, but UDP will be discussed later.

UDPPDUServer is the packet server class that MIK clients talk to, and clients must mix in the **PDUClientInterface** class and implement its upcall, **HandlePDU**, to receive incoming packets. Note that **PDU** is an acronym for Packet Datagram Unit.

Managing data link servers

Before any TCP streams can be created, a data link server must be in place and running. I'll use dial-up PPP as an example for the rest of this chapter, but remember that these concepts apply to any other link server as well. With PPP, the modem must be connected to the remote service provider before the TCP/IP elements of the stack can be started, and the application should also disconnect the modem when it is done. This process of managing the link server is handled automatically by the UsesTCP mixin class in the Magic Internet Kit, but this section will discuss methodologies for handling this if you want to do it by hand.

Source Code Note: The UsesTCP class is defined and implemented in Means:UsesTCP.(Def, c)

Determining if a link server is already in place

Before starting up a link server, it is wise to check and make sure that one is not already in place. The application could keep track of this in a state variable or something, but there is no need to since IP has that functionality already built in. The **FindLinkByClass** method of the IPSwitch class will let the application find out if a particular class of link server is already actively hooked into IP. Here's an example of getting the active link server instance:

```
linkServer = FindLinkByClass(iIPSwitch, linkClass);
```

In this case, linkClass is the class number of the link that you're looking for, e.g. PPPLinkServer_, and linkServer will be the Object ID of the running link server. If this class of link server is not running, the return value of FindLinkByClass will be nilObject.

Stating a new link server

Using UsesTCP to create the link server

The Magic Internet Kit's UsesTCP class provides a method for easily creating link servers: **CreateDataLink** tells a TCP-based means object to connect its link server if it is not already connected. This method, if successful, will return the object ID of the resulting link server object. If this method fails, it will return nilObject.

Note: Applications creating the link server by hand might want to increment the link server's UseCount attribute by one if they don't want the link to be taken down the next time that UsesTCP_DisconnectTCPService detects that the use count is zero.

Creating the link server by hand

If you don't want to use UsesTCP to create the link server for some reason, here's what is going on behind UsesTCP_CreateDataLink. This section is provided for informational purposes; applications should just use CreateDataLink to do this.

How to start a link server depends somewhat on what type of link server it is. A proper Magic Cap link server would inherit from the **CommunicationStream** class and implement the **Connect** method for establishing the connection. Once the hardware is physically connected to the remote host, and the remote host is set up to start the type of data link server that you want, e.g. the terminal server is in PPP mode, it's time to create a new link server and start it up. Here's the code:

```
/* Connect the hardare, assuming that it supports Connect(). */
Connect(hardware, ticket);

/* Set up paramters for new link server object. */
SetUpParameters(&linkParameters.header, newDataLinkServerObjects);
linkParameters.serialServer = hardware;

/* Create the new link server! */
linkServer = NewTransient(linkClass, &linkParameters.header);
```

```
/* If a source IP number is known, let the link server know. */
SetDataLinkSourceIP(linkServer, mySourceIP);

/* Increment the usage count of the link server. */
SetUseCount(linkServer, UseCount(linkServer)+1);

/* Fire up the link server! */
SetEnabled(linkServer, true);
StartDataLink(linkServer);
AttachLink(iIPSwitch, linkServer, DataLinkSourceIP(linkServer));
```

Creating the link server object is quite straightforward using the NewTransient call. After the object is created, its **DataLinkSourceIP** attribute should be set to an IP address if one is to be manually assigned, otherwise set to zero and the link server should have one dynamically assigned. The link server's **UseCount** attribute should also be set to one to keep track of the number of users. This attribute will be used later to determine when the link server should be shut down; when disconnecting a link user, the link user should disconnect the data link if the resulting use count is zero.

Now that the link server object is all ready to go, the last three method calls will start it up and connect it to IP. Note that these methods might fail, and therefore the code above should be surrounded by exception handling code for whatever exceptions might get thrown. For the PPPLinkServer class, a commHardwareError exception will get thrown if PPP negotiation fails. Also keep in mind that connecting the hardware using Connect may throw its own exceptions, so that code needs to be shielded as well in the same way that the equivalent serial connection would. For a more detailed example, see the UsesTCP class.

Note: Applications do not need to create the IPSwitch object manually as it will create itself at package installation time. Don't destroy the IPSwitch object, either!

Destroying a link server

Using UsesTCP to destroy the link server

The **DestroyDataLink** method of UsesTCP is in the logical inverse of CreateDataLink. This method will check to see if the **UseCount** attribute of the link server is zero, and if so, it will take down the link. There is a related method, **AbortDataLink**, that will do a hard take-down of the link server. Applications should not sure this method, though, because AbortDataLink will take the link down regardless of users and will also ensure that the hardware is disconnected. AbortDataLink therefore should only be used by the connection code in UsesTCP.

Destroying the link server by hand

As with creating the link server, there's no reason that applications should have to implement this code since DestroyDataLink does the same thing, but this information is included for the sake of completeness.

As usual, destroying is easier than creating. The only caveat is that the application should check to make sure that it does not destroy a link server that is being used by other streams within the application. This is easily handled by using the UseCount attribute of the server to keep track of the number of users. Here's an example of closing down a link server:

```
linkServer = FindLinkByClass(iIPSwitch, LinkClass(self));
if (HasObject(linkServer))
{
    ulong useCount = UseCount(linkServer) - 1;
    SetUseCount(linkServer, useCount);
    if (useCount <= 0)
    {
        DetachLink(iIPSwitch, linkServer);
        Destroy(linkServer);
    }
}
```

DetachLink will tell IP that the link server is no longer active, and **Destroy** will both call Disconnect on the link server's hardware and destroy the link server object.

Creating and using TCP streams

Creating a TCP stream and initiating a connection

Once the link server and is running and connected to IP, as described in the previous section, the application can create transport layers like TCP on top of it. This is pretty easy to do, and it is similar to setting up any other connection: fill in the blanks of a Means object and pass it to TCPStream's Connect method. Here's what the code looks like:

```
/* means is a DialupIPMeans object, ticket is a TransferTicket */
ReplaceTextWithLiteral(HostName(means), "192.216.22.142");
SetPort(means, 80);

SetMeans(ticket, means);

/* Create the TCPStream object with default parameters */
stream = NewTransient(TCPStream_, nil);
SetStream(ticket, stream);

/* Connect the stream to the remote host */
Connect(stream, ticket);
```

The means object passed into TCPStream_Connect must inherit from DialupIPMeans, and the HostName field must contain the IP address of the destination. For more information on IP addresses and name resolution, see the Resolving host names with DNS section later in this chapter. If the Connect method of TCPStream fails, it will throw a serverAborted exception, so be sure to catch these exceptions.

Parameters can also be passed to NewTransient when creating the TCPStream object. The only interesting parameter is the source port, which may need to be set if required to be within a certain range. This is usually never the case, though, and therefore should be set to zero so that IP will assign a port dynamically.

Note: Multiple streams can be created on the same data link! Just remember to keep track of how many exist by incrementing the link server's UseCount attribute each time a stream is created and decrement the count when they are destroyed. Neat!

Listening for an incoming TCP connection

The above section describes how to initiate an outgoing connection, but with the Magic Internet Kit applications can also listen for incoming connections. The high level MIK framework does not include this functionality built-in just yet since most applications will only want to initiate connections, but the TCPStream interface is in place and fully functional.

To listen for an incoming connection, create the data link server as usual, but replace the connect code with the following:

```
NewTCPStreamParameters parameters;
SetUpParameters(&parameters.header, 0);
parameters.destinationIPAddress = 0;
parameters.destinationPort = 0;

/* port to listen to */
parameters.sourcePort = 80;

stream = NewTransient(TCPStream_, (Parameters*)&parameters);

Listen(stream);
```

In this case, the stream will listen to port 80, the httpd port. The Listen method will immediately return, but the next Read or Write call will block until a remote host connects to the specified port on the device. Therefore any code following the Listen that relies on the connection being active should be after a Read or Write call! For example, if the application calls CountReadPending immediately following the Listen, the Listen will return immediately and the CountReadPending will also immediately return zero, which is probably not what the programmer wants.

Note: Due to the inherently blocking nature of listening for an incoming connection, or doing most other communications work for that matter, this code should **not** be running on the User Actor! See the Multithreading with Actors section of this document for more information on creating and using new threads.

Using a TCP stream

TCPStream objects are used pretty much like any other CommunicationStream subclass; **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 available yet, for example the server has not sent them, Read will block until either all the bytes are received or an error has occurred. 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:

```
count = Read(stream, &buffer, 1);

if (count != 1)
{
    Fail(serverAborted);
}
else /* count == 1, everything is peachy keen */
{
    Unsigned count = CountReadPending(stream);
    if (count != 0)
    {
        if (count > 254) count = 254;
        Read(stream, &buffer[1], count);
    }

    /* Remember that we already read one character 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 occurred, for example the remote host closing the stream. When Read returns, a check is made to make sure that Read returned a character, and if not an exception is thrown. If the character was indeed read okay, the rest of the code will get 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 TCP stream was closed for some reason, or some other error occurred, Write will instead throw a serverAborted exception. For this reason, all calls to Write should be prepared to catch serverAborted exceptions and handle the error case.

Resolving host names with DNS

While IP addresses look neat, what with all those numbers and periods, humans have a hard time remembering them. For this reason symbolic names are often assigned to refer to IP addresses, for example the name "www.genmagic.com" currently refers to the IP address 192.216.22.142. This feature also allows www.genmagic.com to be repointed to a new machine without forcing users to know about the change or the new address. All of this neat functionality is bestowed upon us by the **Domain Name System** (DNS).

While DNS is very useful for resolving host names, also keep in mind that resolving a name takes time; a UDP packet requesting the name resolution must be sent a known name resolver, and then the response packet must be received. Fortunately, DNS caches names once they have been resolved, but the initial lookup still takes a second or two. Additionally, adding the DNS resolver to your Magic Cap application increases the memory footprint by about ten kilobytes.

DNS in the Magic Internet Kit framework

When using the MIK framework, the UsesTCP class will resolve host names when needed. Specifically, if a host name is not a properly formatted IP address, the UsesTCP class will figure that it's a host name that a DNS server can figure out. It will then ask DNS if it can resolve the name and, if not, fail up to the Means classes with a serverAborted exception. The Means classes will then return nilObject for the stream.

To tell DNS which name resolution servers to use, add the 32-bit IP address of each server to the list in the **DNSServers** attribute of the ConnectableMeans object. This list is a FixedList object, so entries can be added with the **AddUniqueElem** method. If the means object is in the application's object instance file, entries can also be added there.

Source Code Note: The Magic Internet Kit's usage of DNS is contained in the UsesTCP class implemented in Means:UsesTCP.(Def, c)

Using DNS manually

If your application is not using the MIK framework, which automatically handles name resolution, you can still use DNS manually. This section will tell you how to do it.

Source Code Note: The class definition for the DNSResolver class is found in the TCP:DNSResolver.Def file.

Preparing for DNS lookups

The first step in preparing to use DNS is making sure that a link server is active. If there's no data link, DNS can't send its request to the name resolvers. See the above section on managing data link servers for more information on that topic. Once the data link is up, the application needs to tell DNS which domain name resolvers to contact. This is done using the **RegisterDNS** method. Here's the code:

```
RegisterDNS(iDNSResolver, 0x12345678);
```

The server should be specified by its 32-bit IP address. If applications need to translate dotted text IP addresses into 32-bit unsigned format, the **MakeIPAddress** method of IPSwitch will do this.

Source Code Note: The **IPAttributeText** class translates IP addresses between dotted text and 32-bit formats. See Means:UsesTCP.(Def, c) for the implementation of IPAttributeText.

Much like the IPSwitch class, the DNS resolver referred to by the **iDNSResolver** indexical does not need to be created or destroyed by the application. This class will automatically create itself at package installation time.

Looking up a symbolic host name

Once DNS has at least one resolver registered and the link server is active, the application can freely request name and address information. To convert a symbolic host name to an IP address, use the **GetHostByName** method of the DNSResolver class. Here's an example:

```
ReplaceTextWithLiteral(hostName, "www.genmagic.com");

address = GetHostByName(iDNSResolver, hostName);

SPrintF(addressString, "%d.%d.%d.%d", (address>>24),
        (address>>16)&0xff, (address>>8)&0xff, address&0xff);

ReplaceTextWithLiteral(hostName, addressString);
```

In the above code, the hostName parameter is a text object containing the name to be looked up, and the return value is the 32-bit IP address. If the name lookup fails, a **DNSLookupFailed** exception will be thrown, so the application should plan ahead and catch this exception.

Keep in mind that TCPStream's Connect method expects the host name to be a dotted text IP address, so the last two lines of code convert the 32-bit value into that format and put the result back into the hostName text object.

Looking up an IP address

The process of resolving an IP address is almost identical to looking up a host name. The **GetHostByAddress** method takes a 32-bit unsigned IP address and returns a text object containing the host name associated with that address. If no name exists a DNSLookupFailed exception is thrown.

Shutting down DNS

The iDNSResolver object will stick around in transient memory when the application is done using it, but applications might want to unregister DNS servers when it is done using these servers. Additionally, if the user has the ability to modify the list of DNS servers that the application should use, a server should be unregistered when the user specifies that it should no longer be used. To do this, call the **UnregisterDNS** method. The parameters for this method are identical to that of RegisterDNS.

Persistence of DNS

The object pointed to by the **iDNSResolver** indexical is located in transient memory. This means that the object will go away whenever transient memory gets blasted, but it will always be recreated. This means that the DNS resolver's cache will be cleared as well as its list of servers that were registered with the RegisterDNS method. As a result, the application should always register its DNS servers before it plans to use them if there is a chance that transient memory could have been reset since the last registration.

More to follow

This chapter of the MIK Programmer's Guide is not yet completed at this time. To find the latest version of this document, which should be more complete by the time you read this, check out the Magic Cap Developer Resources web pages located at:

```
http://www.genmagic.com/Develop/MagicCap/index.html
```

Reference

This section is not yet completed - see our web site for an update

To find the latest version of the Magic Internet Kit documentation, check out the Magic Cap Developer Resources web pages located at:

`http://www.genmagic.com/Develop/MagicCap/index.html`