icras

Guide to Development Tools

For Microsoft Windows

April 24, 2000

Guide to Development Tools for Microsoft Wiindows

(version 4/17/00)

License

Your use of the software discussed in this document is permitted only pursuant to the terms in a software license between you and Icras

Trademarks

Icras, the Icras logo, DataRover, the DataRover logo, DataRover Remote Access Kit, Magic Cap, the Magic Cap logo, and the rabbit-from-a-hat logo are trademarks of Icras, Inc. which may be registered in certain jurisdictions. The Magic Cap technology is the property of General Magic, Inc., and is used under license to Icras, Inc. Microsoft, Developer Studio, Visual Studio, and Visual C++, are all trademarks of Microsoft Corporation.

All other trademarks and service marks are the property of their respective owners.

Limit of Liability/Disclaimer of Warranty

THIS BOOK IS SOLD "AS IS." Even though Icras, Inc. has reviewed this book in detail, ICRAS MAKES NO REPRESENTATION OR WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK. ICRAS SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE AND SHALL IN NO EVENT BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGE, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Some states do not allow for the exclusion or limitation of implied warranties or incidental or consequential damage, so the exclusions in this paragraph may not apply to you.

Patents

The Magic Cap software is protected by the following patents: 5,611,031; 5,689,669; 5,692,187; and 5,819,306. Portions of the Magic Cap technology are patent pending in the United States and other countries.

United States Government Restrictions

This product is "commercial item" as that term is defined at 48 C.F.R. 2.101 (OCT 1995) consisting of "commercial computer software" and "Commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (SEPT 1995) and is provided to the U.S. Government only as a commercial end item. Consistent with 46 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire this product only with those rights set forth therein.

Icras, Inc.

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

Table of Contents

1

Introduction

A Magic Cap **software package** is a collection of objects that perform a specific set of functions. Magic Cap itself provides several built-in packages, including the datebook, the name card file, the notebook, and the mail package. In addition, Magic Cap provides a fertile environment to support third-party packages. This guide describes how to use Magic Developer, a software development environment for creating Magic Cap packages.

In this guide, we will discuss the process of developing packages for Magic Cap. To use it effectively, you should be familiar with the following:

- C++ programming language
- Object-oriented programming
- Windows software development
- Operating a Magic Cap DataRover 840

Developing software packages

Software for Magic Cap is distributed in packages that contain objects for performing tasks. Some packages are conventional applications with specific purposes, such as electronic mail and personal finance. Other packages are sets of objects required by a variety of applications in Magic Cap, such as user-interface features like buttons and clocks.

Packages can vary according to their purpose and structure. Some special purpose packages contain code to implement specific features, such as an inventory checker. Other packages might not contain any code—they simply add objects to Magic Cap. This second category of packages can include data objects such as stamps and sounds, as well as other packages that benefit from Magic Cap's rich set of user-interface tools. Other packages may fall anywhere between these examples.

Microsoft Visual C++

Magic Developer is based on Microsoft Visual C++, an application development environment for Windows software. You must install either the Professional Edition or the Enterprise Edition of Microsoft Visual C++. Magic Developer extends the Microsoft Visual C++ environment with tools for building and testing Magic Cap packages. See Chapter 2, “Building Software Packages,” for more information.

Magic Cap Simulator

While developing Magic Cap packages, you’ll use a version of Magic Cap running on the PC called the Magic Cap Simulator that lets you create, edit, and specify the behavior of live objects graphically. This allows you to see and use packages much as they will appear on the DataRover 840.

Magic Cap Simulator simulates a DataRover 840 and allows you to develop software without having to move your package to an actual DataRover 840 every time you make a change to your package, saving time in the development cycle.

Magic Cap Simulator is useful for two purposes. You can use it to run and test packages with greater convenience than downloading the package into a DataRover 840. In addition, you can use it in the software development process to modify objects and then dump them back as ASCII text in object definition files.

See Chapter 3, “Magic Cap Simulator,” for more information on how to use Magic Cap Simulator to create and modify objects for your packages.

Construction tools

Typically, you’ll start creating your package in Magic Developer, then build it and run it in the Magic Cap Simulator. In the Magic Cap Simulator, you can use the tools and techniques available in **construction mode** to modify your package. In construction mode, you can create new viewable objects by dropping them from the Magic Hat, and then modifying them with the move, copy, and stretch tools. See “Construction tools” on page 37 for complete information about construction mode.

Inspector

The Magic Cap Simulator includes an object analyzer called the Inspector that you use to examine objects at run-time. The Inspector lets you view the content of objects and dump them to the log file. See “Inspector” on page 52 for more information.

Bowser Jo

The complete collection of classes for developing Magic Cap packages is very extensive. To help you navigate through these classes, the Software Development Kit (SDK) includes Bowser Jo, a tool for viewing the Magic Cap class hierarchy that runs in your web browser. See Chapter 4, “Bowser Jo,” for more information.

Debugging tools

Part of developing a Magic Cap package is the necessary process of finding and fixing bugs. Magic Developer provides two debugging environments for Magic Cap package development:

- Microsoft Visual C++ and the Magic Cap Simulator
- The GNU source-level debugger (GDB) and the DataRover 840

See Chapter 5, “Debugging Tools,” for more information on how to use the debugging tools to develop Magic Cap packages.

Object Tools

Object Tools translate text descriptions of Magic Cap objects into live, graphical representations of the objects. The Magic Cap Simulator Object Tools can also reverse this operation, converting live objects back to their text representations.

Object Tools process two kinds of files, and most packages include at least one file of each kind (in addition to one or more source files that are processed by conventional compilers and assemblers). The first kind contains the descriptions of any classes defined by the package; this is called a **class definition file**. The second kind, called the **instance definition file**, contains descriptions of objects defined by the package.

See Chapter 6, “Object Tools,” for more information on how to use Object Tools to develop Magic Cap packages.

Magic Script

Magic Cap uses a simple but powerful scripting language called **Magic Script** to provide a high-level way of arranging and connecting objects. Magic Script uses a Java based model of execution. When you create your own packages, you may write and edit Magic Script in any object definition file. For an example, see the sample package TicTacToe.

Magic Script is most useful in coordinating user-interactions with viewable objects like buttons. It is easier to write small scripts for these than to create the equivalent object subclasses. From a performance perspective, the two approaches are equivalent. See “Magic Script” on page 98 for more information.

Localization tools

In addition, Magic Developer has tools for localizing Magic Cap packages for different languages.

See Chapter 7, “Package Localization,” for more information on how to use these localization tools to develop localized versions of Magic Cap packages.

2

Building Software Packages

This chapter describes how to build packages with Magic Developer. First we'll give you an overview of the package development process. Then we'll show you how to build a simple Magic Cap package named HelloWorld, giving you a whirlwind tour of the Magic Developer development environment. From there we'll get more specific about the different components that go into Magic Developer and what they do.

Magic Developer contains a number of sample packages in addition to HelloWorld. Look in the Samples folder in Magic Developer to see these packages. You can learn something about them by looking at their source files and then building and running them.

Overview of package development

The Magic Cap Simulator provides an environment for constructing and testing packages on your PC. The fastest way to develop packages is to take advantage of the simulator, as follows:

1. Create a package using the simulator as the target.
2. Run the package in the simulator to construct it.
3. Debug the package in Microsoft Visual C++.
4. After the package is working, create a build using the DataRover 840 as the target.
5. Download the package to the DataRover 840 and finish debugging it there.

Magic Developer uses the standard Microsoft Visual C++ compilers and linkers to build packages, as well as a set of tools called Common Tools (common across all platforms and targets). The common tools include the Class Compiler, Object Compiler, X-File Linker, and the Package Builder.

When you target the Windows simulator, the code and data for a package are built into a dynamic link library (DLL). The Common Tools produce a frozen package which is loaded along with the DLL. When you target any other platform, the code and data are stored inside the frozen package file. See “Package Formats” on page 21 for more information about frozen packages.

Building a simple package

Let’s clone one of the sample packages delivered with Magic Developer and use the clone to learn how to build a package. Cloning an existing package is usually a good way to create a new package because the required source files are copied to a new directory for you.

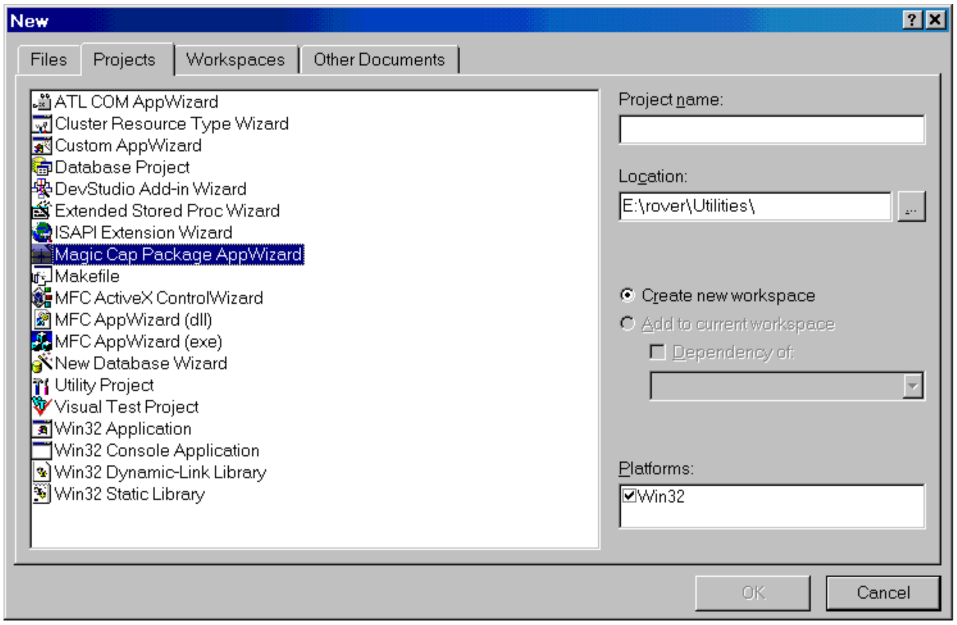
How to clone a package

Here’s a list of steps for cloning a very simple Magic Cap package.

1. Launch Microsoft Visual C++ and choose File▶New.

   The New dialog box appears.

2. Select the Projects tab, if it is not already selected.



Project name:

Location: E:\rover\Utilities\

● Create new workspace

○ Add to current workspace

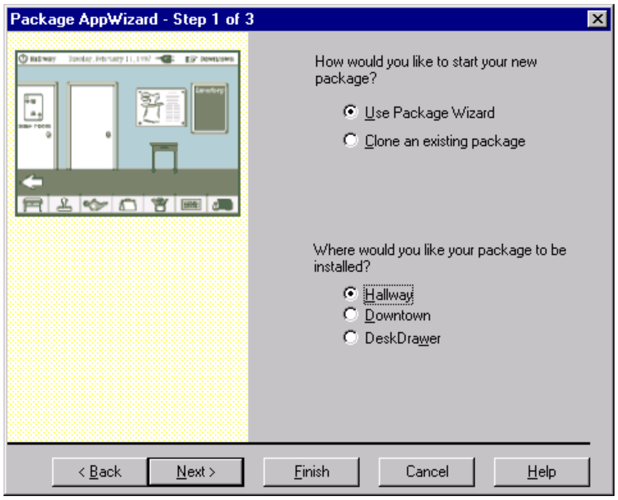
□ Dependency of:

Platforms: ☑ Win32

3. Enter the following values, then click OK:

Option	Value
`Left Panel`	**Magic Cap Package AppWizard**
`Project name`	**MyHelloWorld**
`Location`	**The samples subdirectory under the directory in which Magic Developer is located.**

The Package AppWizard appears.



How would you like to start your new package?

● Use Package Wizard

○ Clone an existing package

Where would you like your package to be installed?

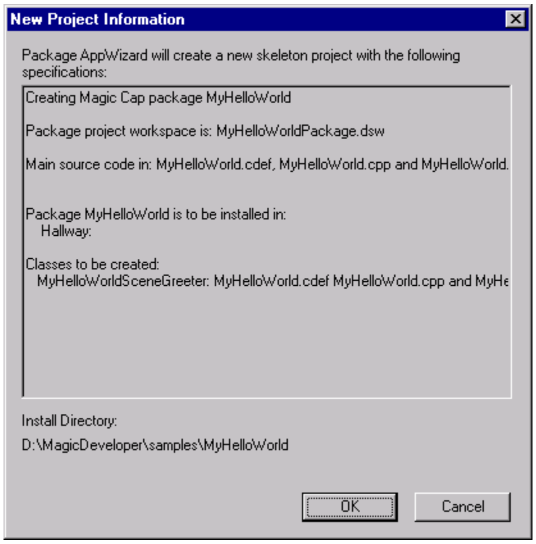
● Hallway

○ Downtown

○ DeskDrawer

4. Check Clone an existing package, then select HelloWorld in the Packages list and click Finish.

   The New Project Information dialog box appears.

   Package AppWizard will create a new skeleton project with the following specifications:

   Creating Magic Cap package MyHelloWorld

   Package project workspace is: MyHelloWorldPackage.dsw

   Main source code in: MyHelloWorld.cdef, MyHelloWorld.cpp and MyHelloWorld.

   Package MyHelloWorld is to be installed in:
   Hallway:

   Classes to be created:
   MyHelloWorldSceneGreeter: MyHelloWorld.cdef MyHelloWorld.cpp and MyHe

   Install Directory:
   D:\MagicDeveloper\samples\MyHelloWorld

5. Click OK.

   The MyHelloWorld workspace opens.

6. Choose File▶Close Workspace and close the MyHelloWorld workspace without saving it.

   You actually use the MyHelloWorldPackage workspace to build MyHelloWorld, so this workspace is not necessary.

7. If the Microsoft Visual C++ Workspace Viewer is open, close it.

   You cannot open files in a Magic Developer project from the Microsoft Visual C++ Workspace Viewer; instead, you use the Magic Developer Package Viewer. See “Displaying and managing the contents of a package” on page 61.

How to create a new package

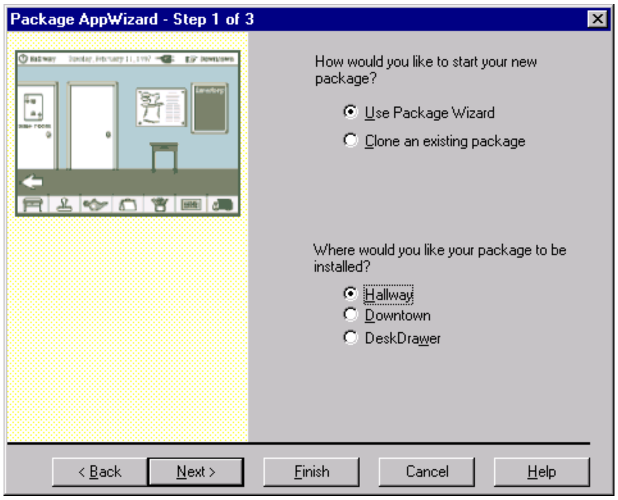
Here's a list of steps for creating a new Magic Cap package without cloning

1. Launch Microsoft Visual C++ and choose File▶New.

   The New dialog box appears.

2. Select the Projects tab, if it is not already selected.
3. Enter the requested values as you do when cloning a package, then click OK.

   The Package AppWizard appears.



Package AppWizard - Step 1 of 3

How would you like to start your new package?

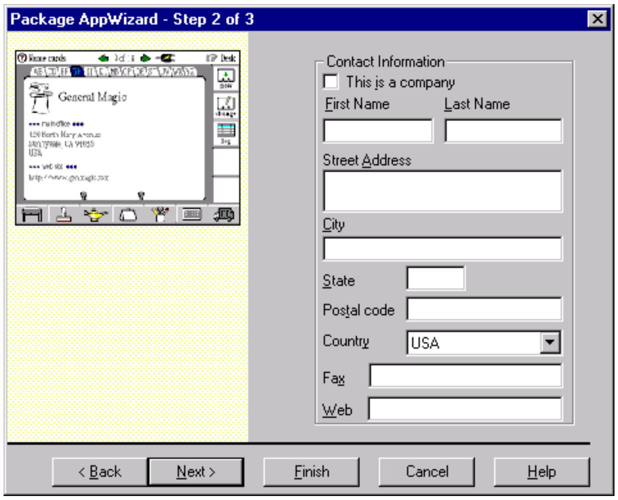
● Use Package Wizard
○ Clone an existing package

Where would you like your package to be installed?

● Hallway
○ Downtown
○ DeskDrawer

4. Check Use Package Wizard, then specify the location of the main package entry point and click Next.

   The following dialog box appears.



Package AppWizard - Step 2 of 3

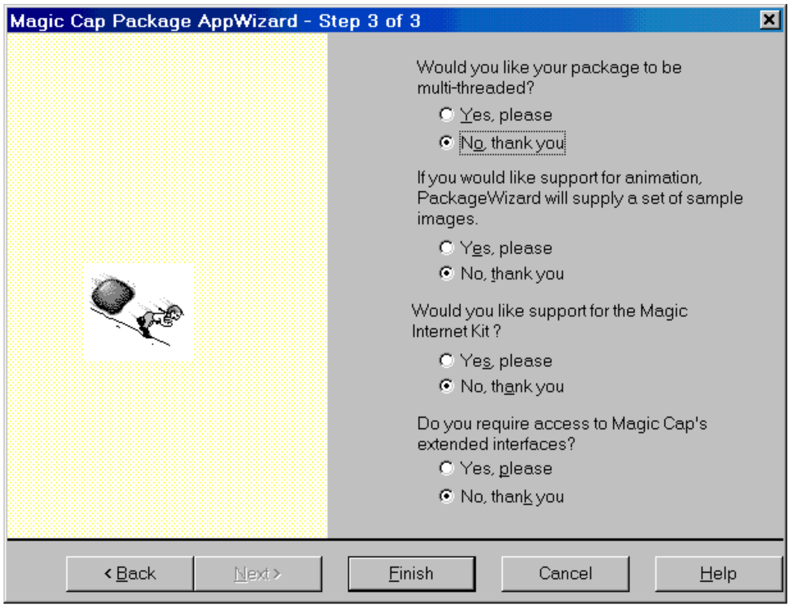
Contact Information

□ This is a company

First Name
Last Name
Street Address
City
State
Postal code
Country: USA
Fax
Web

5. Enter contact information for users, then click Next.

   The following dialog box appears.



Magic Cap Package AppWizard - Step 3 of 3

Would you like your package to be multi-threaded?

○ Yes, please
● No, thank you

If you would like support for animation, PackageWizard will supply a set of sample images.

○ Yes, please
● No, thank you

Would you like support for the Magic Internet Kit ?

○ Yes, please
● No, thank you

Do you require access to Magic Cap's extended interfaces?

○ Yes, please
● No, thank you

6. Specify whether you want to include support for a multi-threaded package, animation, or the Magic Internet Kit, then click Finish.

   Note that the last option which allows you to select access to extended interfaces is an advanced option in most cases should not be selected. Ensure that the selection is set to No.

   The New Project Information dialog box appears.

7. Click OK.

   Microsoft Visual C++ creates a set of files you can use as a starting point for creating a new package, and a new project workspace called `*PackageName*.dsw` opens.

8. Close the workspace as you do when cloning a package, and open the workspace called `*PackageNamePackage*.dsw`.

9. If the Microsoft Visual C++ Workspace Viewer is open, close it also.

   You cannot open files in a Magic Developer project from the Microsoft Visual C++ Workspace Viewer; instead, you use the Magic Developer Package Viewer. as described in “Displaying and managing the contents of a package” on page 61.

How to build a package

When you build a package, you need to specify the following information:

- The device upon which the build will run.

  Win32 targets the Magic Cap Simulator.

  Apollo targets the DataRover 840.

- The type of build.

  Debug builds contain full symbolic information for debugging purposes.

  Release builds are smaller and faster than debug builds because they do not contain symbolic information, which means you cannot use them for debugging.

- The country in which the build will be used.

  USA and **Japan** are both supported in this release of Magic Developer. See Chapter 7, “Package Localization,” for more information.

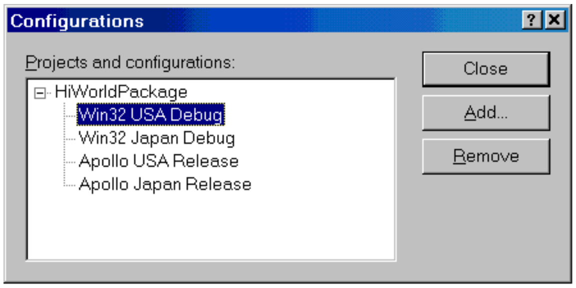
Let’s create a debug build of the cloned HelloWorld package, targeting the Magic Cap Simulator. Here’s a list of steps for building your cloned package.

1. Choose File▶Open Workspace, navigate to the MyHelloWorld subdirectory underneath MagicDeveloper\samples, then select MyHelloWorldPackage.dsw and click Open.

   The MyHelloWorldPackage workspace is now open in Microsoft Visual C++.

2. Choose Build▶Set Active Configuration to specify the device target, the type of build, and the localization country.

   The Configurations dialog box appears.



Projects and configurations:

HiWorldPackage
- Win32 USA Debug
- Win32 Japan Debug
- Apollo USA Release
- Apollo Japan Release

3. Choose MyHelloWorldPackage—Win32 USA Debug and click OK.

   The Set Active Configuration dialog box closes.

4. Choose Build▶Build MyHelloWorld.dll or press *F7*.

   Microsoft Visual C++ builds the package.

How to test a package

You can test the MyHelloWorld package directly within Microsoft Visual C++ by entering the Debugger. To test your newly cloned package:

1. The first time you test a package it will be necessary to establish the executable and package execution environment. You will only have to do this one time:

   a. If you have not already done so, ensure that the default configuration for your package is "Win32 USA Debug". If this is not the default configuration, select the Build ▶ Set Active Configuration and set the default as specified.

   b. Select Project ▶ Settings and click the Debug tab.

   c. In the "Executable for debug session" text box you need to point to the Magic Cap Windows Simulator. Do so by clicking the button on the right and navigating to:

   <installation directory>\debug\win32\MagicCap-USA.exe

   where <installation directory> is where you installed MagicDeveloper.

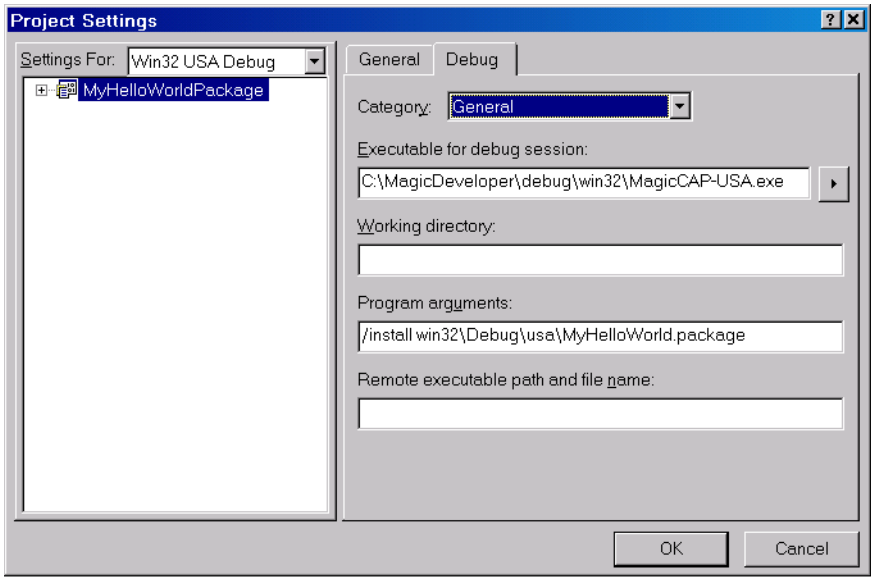
   d. In the "Program arguments" text box you need to tell the simulator where to find your package image file. Do so by typing the following:

   /install win32\debug\usa\<package name>.package

   where <package name> is the name of your package, and then click OK. For example, assuming that you created a package named MyHelloWorld, the command line would be:

   /install win32\debug\usa\MyHelloWorld.package

   e. Select File ▶ Save Workspace to save your changes. Below is a sample of the result when MagicDeveloper is installed in C:\MagicDeveloper.



Project Settings

Settings For: Win32 USA Debug

MyHelloWorldPackage

Category: General

Executable for debug session: C:\MagicDeveloper\debug\win32\MagicCAP-USA.exe

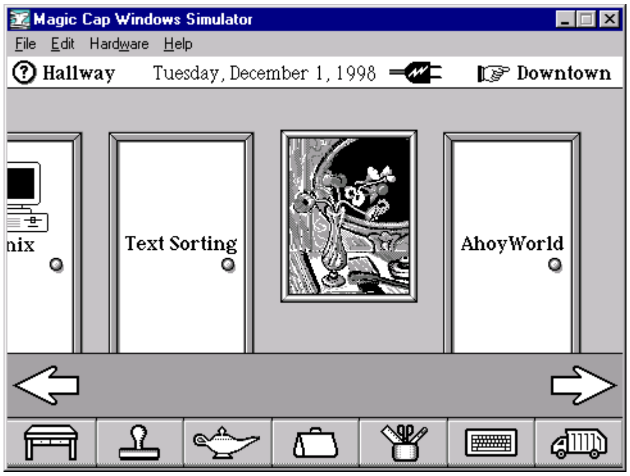
Working directory:

Program arguments: /install win32\Debug\usa\MyHelloWorld.package

Remote executable path and file name:

2. Choose Build ▶ StartDebug ▶ Go or press *F5*.

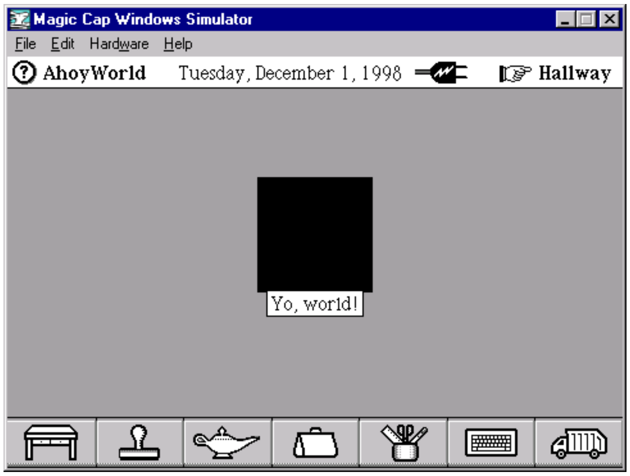
   Microsoft Visual C++ launches the Magic Cap Simulator and loads your package. The MyHelloWorld package appears as a door labelled AhoyWorld in the Hallway.



Note: If you are unable to run your package, see “Troubleshooting” on page 20.

3. Click the AhoyWorld door.

   The Magic Cap Simulator displays a black box labelled “Yo, World.” The black box plays a sound when you touch it.



4. Choose File ▶ Exit in the Magic Cap Simulator to close the simulator and the package.

See Chapter 5, “Debugging Tools,” for more information on debugging and testing.

Troubleshooting

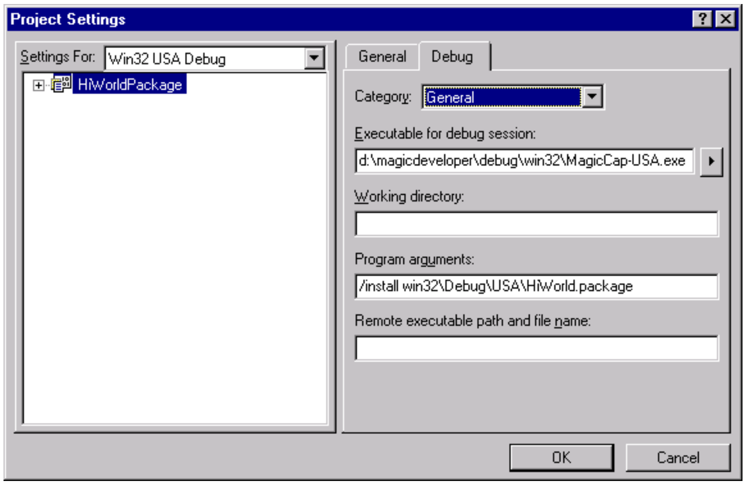
If you are unable to run a package using the information in “How to test a package” on page 18, or if Microsoft Visual C++ asks you for the name of an executable to load, your debug project settings are not correct. Follow these instructions to set up your project:

1. In Microsoft Visual C++, choose Project ▶ Settings.

   The Project Settings dialog box appears.

2. Select the Debug tab.

   The Debug panel appears.



Settings For: Win32 USA Debug

HiWorldPackage

Category: General

Executable for debug session: d:\magicdeveloper\debug\win32\MagicCap-USA.exe

Working directory:

Program arguments: /install win32\Debug\USA\HiWorld.package

Remote executable path and file name:

3. Enter the following value in the *Executable For Debug Session* field:

   ```
   %SDKROOT%\debug\win32\MagicCap-USA.exe
   ```

   where `%SDKROOT%` represents the name of the directory in which you installed Magic Developer.

4. Enter the following value in the *Program Arguments* field:

   ```
   /install win32\debug\usa\PackageName.package
   ```

You should now be able to run your package under the debugger as described in “How to test a package” on page 18.

Package Formats

Magic Cap packages have a specific format called the Frozen Objects Format. This format divides a package into sections or attributes, containing the following pieces:

- Code
- Global data
- Package class and object definitions
- Exported interfaces (classes/operations/indexicals)
- Imported interfaces (classes/operations/indexicals)

Simulator debug packages

For debug packages which target the simulator, Magic Developer builds the code into a DLL. To facilitate debugging, simulator debug packages do not contain the DLL; instead, they contain a place holder within the code attribute that tells the simulator to load the DLL from the same directory.

Because the DLL and package are separate, you typically must deliver two files to distribute a simulator debug package:

- *`PackageName`*`.package`
- *`PackageName`*`.DLL`

To build a simulator debug package as a single file, remove the `-separate-library` switch from the command line passed to the `BuildPackage` command

Simulator release packages

For release packages which target the simulator, Magic Developer also creates a DLL; however, the DLL is built into the code attribute. At runtime, the simulator writes the DLL into a temporary file and loads it from there. Since the DLL contains both the code and the global data for the package, simulator packages do not have a global data attribute.

MIPS packages

For packages which target a MIPS platform, Magic Developer builds the code into an ELF executable. The code and data is extracted from this executable and placed in the appropriate frozen package attributes. Consequently, a DataRover 840 package is always a single file (*`PackageName`*`.package`) which you can download or mail to a device.

Inside a Magic Cap package

Let's examine the contents of the folder `MyHelloWorld`. You will find the following four files:

- `Objects.odef`. This is the object definition file. It defines the static objects for the package.
- `MyHelloWorld.cdef`. This is a class definition file. It defines the classes that are unique to the package. MyHelloWorld defines a single class named `Greeter`.
- `MyHelloWorld.cpp`. This file contains C++ source code to the `Greeter_Draw` method for the package's `Greeter` class.
- `MyHelloWorld.make`. This is a Microsoft Visual C++ build script that is generated automatically.

In addition, you will find the following sample phrase files used for localizing the package (see the localization section in the Magic Cap SDK documentation for more information):

- `France.Custom.Phrases`
- `France.Package.Phrases`
- `Japan.Custom.Phrases`
- `Japan.Package.Phrases`
- `Japan.Resizing.Phrases`
- `USA.Custom.Phrases`
- `USA.Package.Phrases`

Object instance definition file

The file `Objects.odef` contains a series of static object instance definitions that the Object Compiler uses to build the objects for a Magic Cap package. Object instance definition files use the `.odef` suffix.

Your package must include at least one object instance definition file to be compiled by the Object Compiler.

Here’s an example of an instance definition taken from `Objects.odef` in the `MyHelloWorld` folder:

```
instance SoftwarePackageContents contents 'MyHelloWorld';
           dateCreated: 0;
           timeCreated: 0;
          dateModified: 0;
          timeModified: 0;
          autoActivate: true;
      installationList: (ObjectList installationList);
                author: iGeneralMagic;
             publisher: iGeneralMagic;
           versionText: nilObject;
         helpOnObjects: (ObjectList helpForObjects);
    sceneIndexicalList: nilObject;
    stackIndexicalList: nilObject;
          startupScene: nilObject;
           startupItem: nilObject;
          creditsScene: nilObject;
                  logo: iTurkey;
responseCardStationery: iDefaultStationery;
                hidden: false;
        dontDeactivate: false;
end instance;
```

Each instance definition includes three parts:

- The instance header—a single line that consists of the keyword `instance` followed by the name of the object’s class, followed by a symbolic tag that must be unique for this package, followed by an optional object name which may contain spaces and is enclosed by single quotes, ending with a semicolon. The unique tag for `SoftwarePackageContents` must be `contents`.
- The body with one or more lines that list the object’s fields and their values.
- The instance footer—a single line that consists simply of the keywords `end instance;`

A typical instance definition file will have a series of object instance definitions See “Instance definition file” on page 92 for more information about instance definition files and their syntax.

Class definitions

If your package includes any new classes, as most packages do, you must define them in a class definition file to be compiled by the Class Compiler. A class definition file is a text file that consists of one or more class definitions. Typically, a package’s class definition file has the same name as the package, plus a `.cdef` suffix.

Here’s an example of a class definition taken from the class definition file in the MyHelloWorld package, followed by a discussion of its contents:

```
define class Greeter;
  inherits from Viewable;
  overrides Draw;
end class;
```

This class definition includes four parts:

- The class header—one or more lines that consist of the keywords `define class`, followed by the name of the new class, optionally followed by other information about the class.
- The superclass designation—the keywords `inherits from` followed by the name of the class's immediate superclass.
- The body—one or more lines that list the class's fields, operations, and attributes. Overridden methods are specified with `overrides`.
- The class footer—a single line that consists of the keywords `end class;`

See "Class definition file" on page 83 for more information about class definition files and their syntax.

Source code

If your package includes any new classes, you must define the code that implements the methods of the new classes in C++ files. You can arrange your source code in any collection of files. Microsoft Visual C++ and Magic Developer require the `.cpp` suffix.

If a package has only one source file, it usually has the same name as the package plus a `.cpp` suffix. Magic Developer requires that your package have at least one `.cpp` file; if your package has no code, this file may just be an empty file.

Following is an example of a C++ source file from the MyHelloWorld package:

```
#include "MagicCap.h"
#include "Debug.h"

#include "MyHelloWorld.xh"
#include "MyHelloWorld.xph"

#undef CURRENTCLASS
#define CURRENTCLASS Greeter

/* ---------------------------------------------------------------------
 Here's the Draw() routine, which only needs to draw the content area, in
 this case, it just fills a box. This routine is called by the system when
 our object needs to be drawn.
*/

Method void
Greeter_Draw(Reference self)
{
   Box     ourContentBox;
   ulong   color;

   /* Get an rgb color value from our viewable */
   color = PartColor(self,
                     Highlighted(self) ? partAltContent : partContent);
   /*
   Now, get contentBox and pass it and the color we want and the transfer
   mode to FillBox().
   */
   ContentBox(self, &ourContentBox);
   FillBox(CurrentCanvas(), CurrentClip(), &ourContentBox, color,
           pixelDither | pixelCopy);
}
```

```
#undef CURRENTCLASS
```

Although most of this file is written in standard C++, you should notice these vitally important special elements:

- This file uses `#include` statements to include a number of files provided with Magic Developer; see the sample packages for the exact list of files to be included. You might also have your own header files that would be referenced with an include statement.
- The file includes a statement that specifies the class (`#define CURRENTCLASS Greeter`). This statement is required, and the class name must match your class name exactly, including case. If they don't match, your package may not compile. If you define more than one class, make sure that you set the class name correctly for each method. You can have as many of these class specifiers as you need, switching from one class to another before each method if necessary.
- Each function declaration for a method begins with the keyword `Method`.
- In Magic Cap, each method's name is the name of the class, followed by an underscore, followed by the operation name (as defined in the `operation` statement in the class definition file). The Class Compiler uses this convention to match operations to C++ functions. Of course, your source code can have functions that are not methods, and there are no naming restrictions on them.
- The first parameter to every object method is an object reference, and the first parameter to every class method is a class reference. For convenience, this parameter is never declared in the class definition file. However, it must be included when you declare the methods in the source files, or the compiler won't know about it. See "About the object reference" on page 68 for more information.

Makefile

Microsoft Visual C++ uses the **nmake** utility to organize the software build process. The `make` description file, or Makefile, contains a list of instructions for building a software module. Magic Developer uses this system for building Magic Cap packages.

The makefile contains Microsoft Visual C++ commands that build your package. If you create a new package by cloning an existing package, Microsoft Visual C++ creates a makefile that you can use as a template. You can modify this makefile, if necessary, as you add files to your project. See "Customizing makefiles" on page 26 for more information.

Customizing makefiles

The makefiles generated by the package wizard are suitable for applications which do not include libraries or which do not import interfaces from other packages. If you wish to do any of these things, you will need to customize your makefile.

Default makefile

The makefiles generated by the package wizard, such as all the sample makefiles, are created in the following form:

```
# Magic Developer Generated Makefile
#    ** DO NOT EDIT **

PACKAGE_NAME = Temp

CDEF_X_FILES = \
$(INTERMEDIATE_DIR)\Temp.cx \
$(INTERMEDIATE_DIR)\TempIndexicals.cx \
#

ODEF_X_FILES = \
$(INTERMEDIATE_DIR)\Objects.ox \
#

CPP_O_FILES = \
$(INTERMEDIATE_DIR)\Temp.o \
#

!include $(SDKROOT)\scripts\packagewizard\MakePackage
```

Defining variables

Each package makefile defines a number of variables which are used by the `MakePackage` file to build the package. For example, you can define variables that specify a directory to search for .cdef.x/.h files as follows:

```
!ifdef  DEBUG
     CAP_TOOL_INCLUDES= -I ..\ExportSample\$(TARGET)\Debug
!else
     CAP_TOOL_INCLUDES= -I ..\ExportSample\$(TARGET)\Release
!endif
```

In the previous example, the `ImportSample.make` makefile (in the `samples` directory under the directory in which Magic Developer is installed) imports an interface from the ExportSample package, and therefore needs to include `ExportSample.cdef.x` from the `ExportSample` directory.

You can also use variables in a makefile to specify which C++ files will be built with precompiled headers:

- All the files in `CPP_O_FILES` are built with precompiled headers. These files have the extension `.o`.
- All the files in `CPP_O_FILES_NP` are built without precompiled headers. These files have the extension `.no`.

By default, the package wizard puts all the files in the `CPP_O_FILES` group. You must move any files you want to build without precompiled headers to the other group.

For more variables you can define in your makefiles, see the comments in the `MakePackage` file in the `scripts\packagewizard` directory under the directory in which Magic Developer is installed.

Using the Magic Internet Kit

The Magic Internet Kit is a complete development kit for creating communicating Magic Cap applications. You can customize your makefile to provide access to the Magic Internet Kit by adding the following line:

```
USE_MAGIC_INTERNET_KIT = 1
```

The CujoTerm sample application provides an example of using the Magic Internet Kit.

For more information about the Magic Internet Kit, refer to the guide entitled, *“Magic Internet Kit”*.

Extra dependencies

It is also possible to specify extra dependencies for a makefile. For example, the `ImportSample.make` makefile in the `samples` directory under the directory in which Magic Developer is installed uses extra dependencies.

The ImportSample package depends on ExportSample because a build of ImportSample for a specific target must be conditioned by a build of ExportSample for the same target. This relationship is expressed in the makefile as follows:

```
EXTRA_DEPENDENCY = ..\
        $(SDKROOT)\samples\ExportSample\$(LOCALE)\ExportSample.package
#

$(SDKROOT)\samples\ExportSample\$(LOCALE)\ExportSample.package:
        cd $(SDKROOT)\samples\ExportSample
        nmake /f MakeExportSamplePackage.mak all
        cd $(SDKROOT)\samples\ImportSample
```

In this example, the following lines specify that the ExportSample package is a dependency of ImportSample:

```
EXTRA_DEPENDENCY = ..\
        $(SDKROOT)\samples\ExportSample\$(LOCALE)\ExportSample.package
#
```

The following lines in this example define the rule to build the ExportSample package:

```
$(SDKROOT)\samples\ExportSample\$(LOCALE)\ExportSample.package:
        cd $(SDKROOT)\samples\ExportSample
        nmake /f MakeExportSamplePackage.mak all
        cd $(SDKROOT)\samples\ImportSample
```

Modifying viewable objects with construction tools

Objects that make a visual appearance are called **viewable objects**. They are the visible building blocks of the user-interface. After you build and run your package with Magic Cap Simulator, you'll want to modify the package's appearance and store the results back into the package's source code.

Modifying viewable objects has two purposes: to refine the visual appearance of the elements of a user-interface, and to assist in localization.

You will use the Magic Cap Simulator to modify viewable objects. Here's how to do it:

1. Launch the Magic Cap Simulator and the MyHelloWorld package from within Microsoft Visual C++.

   You can launch the Magic Cap Simulator together with a package you've built by choosing Build▶StartDebug▶Go or pressing *F5*. This runs the package in the current workspace, as described in "Building a simple package" on page 12.

2. Turn on **construction mode**.

   a. Go to the Hallway, find the Controls door and click it.

   b. Click *general*.

      Two groups of check boxes appear.

   c. Turn on the check box for *construction mode*.

      The Stamper icon at the bottom of the Magic Cap screen changes to a Magic hat icon to indicate that you are in construction mode. This means that some Magic Hat objects have extra features, particularly for changing the appearance of other objects.

3. Return to the MyHelloWorld package.

   a. Click the *step-back pointer* in the top right corner of the Magic Cap screen.

      The Controls scene appears.

   b. Click the *step-back pointer* again.

      The Hallway appears.

   c. Go down the Hallway to the door named AhoyWorld, then click it to enter.

      The AhoyWorld scene appears.

4. Create a **color coupon** and apply it to a viewable object.

a. Click the Magic Hat at the bottom of the screen.

b. Click the category box named *colors*.

A series of different shades will appear for you to select from.

c. Select a color and apply it to a viewable object.

When you click a color to select it, the Magic Hat window closes and the color remains in a special viewable called a **coupon**. You can apply this coupon to another viewable object by dragging the coupon over the object and releasing it. Magic Cap changes an object's color to indicate when you are over an object which can accept the color coupon.

5. Dump the modified objects from the Magic Cap Simulator.

a. Choose Examine▶Dump Package in the Magic Cap Simulator.

Note: If you don't see Examine in the menu bar, choose File▶Show Development Tools. That should display extra menu items used for development.

A dialog box lets you specify the location and name of a file.

b. Specify the location and name of the dump file, then click Save.

c. Quit the Magic Cap Simulator and return to Microsoft Visual C++.

6. Import the modified objects into Magic Developer by doing either of the following:

- Delete `Objects.odef` and rename the dump file to `Objects.odef`.
- Open the dump file and `Objects.odef`, then copy each object instance from the dump file into `Objects.odef`.

This creates a new version of your instance definition file that includes the color change for the Greeter object. Make sure you merge any comments from your original `.odef` file into your new file.

7. Save the files and rebuild the package.

When you do this you should notice that the box in the MyHelloWorld scene has the new color. You can continue to modify your package by changing the source or by rebuilding and returning to the Magic Cap Simulator at any time.

By dumping from the Magic Cap Simulator and then importing dumped objects to Magic Developer, you can move back and forth between the two environments.

When using the Magic Cap Simulator, you can do a lot more than just modify existing objects. You can add new objects from the Magic Hat. Then, when you dump and import the dumped objects, the newly added objects will be in your instance definition file.

You can also dump individual objects that you create or modify in the simulator to merge into `Ojects.odef`. See “Object dumping” on page 93 for more information.

Downloading a package to a storage card

To install your package onto a device, you normally download it to a storage card in a DataRover 840. The package is stored on the card as a raw byte stream that can be read by the DataRover 840 during either a cold or a warm boot.

Optionally, you can use the WinPCLink tool to download a package to a storage card. Refer to the WinPCLink documentation for instructions.

How to perform a cold boot

You need to perform a cold boot of your DataRover 840 to download a package and to perform other tasks such as debugging with GDB. Follow these steps to perform a cold boot of your DataRover 840:

1. Turn off the device, remove both batteries, and unplug the power cord and the serial cable.
2. Wait a few seconds to make sure the memory is reset.
3. Plug in the power cord and the serial cable, replace the batteries, then turn on the device.

How to download a package

Use the following steps to download a package to a storage card:

1. In Magic Developer, build your package with an Apollo build target.

   Magic Developer creates both the package and also the file `download.bat` that you use to download the package.

2. In the Control Panel of your PC, confirm that the `MagicPort` environment variable correctly identifies the serial port you are using.

   - In Windows NT and Windows 2000, set the environment variable in the Control Panel.
   - In Windows 98, set the environment variable in the `autoexec.bat` file.

   For example, if the DataRover 840 cable is attached to com2 on your PC, the value of `MagicPort` should be “com2”.

3. Place a storage card in the DataRover 840 to receive the package.
4. Enter Monitor mode by cold booting while holding down the *Option* button.

   The screen remains blank; however, the device is now on in Monitor mode.

5. Make sure that the cable connects the serial port of your PC to the Magic Bus port of the DataRover 840.

6. Open a command prompt window, change to the directory that contains your package, then run the batch file `download.bat` to copy the package to the card in your DataRover 840.

   The screen displays messages such as the following:

   ```
   # writing 0x464BC bytes at 0x24000000, estimated time: 44.99 seconds
   # transfer completed, checking errors...
   # actual time: 33.23 seconds
   ```

7. Cold boot the DataRover 840 without holding down the *Option* button.

   Magic Cap reads the package into main RAM and displays the message “Do you want to set up the storage card...”

8. Do either of the following:

   - Choose *don’t* if you want to use the same card to load the package onto other devices.
   - Choose *set it up* if you want to use the card for other purposes.

9. After the cold boot has completed, power down the device and remove the storage card.

   If you do not remove the card, the package is read into main RAM again each time you boot.

Downloading multiple packages

Occasionally you develop multiple packages which work together, such as the ExportSample and ImportSample packages in the `samples` directory. When you are working in the simulator, you can load both packages manually in the required order. When you are downloading them to a DataRover 840, however, you must concatenate the packages so they load onto the device at the same time.

To concatenate packages, use the `concat` utility provided with Magic Developer. For example, to concatenate the ExportSample and ImportSample packages:

```
concat ..\ExportSample\apollo\Release\USA\ExportSample.package
     apollo\Release\USA\ImportSample.package x
move x apollo\Release\USA\ImportSample.package
```

You may then download the combined import sample package file to the DataRover 840 using the download batch file described in “How to download a package” on page 30.

3

Magic Cap Simulator

The Magic Cap Simulator is a development version of Magic Cap that runs as a Windows application. It has special features that are not available in the version of Magic Cap that runs on DataRover 840s. Most of these features are available from items in the simulator menu bar. Others, like construction tools, are available within the Magic Cap Simulator.

Magic Cap Simulator menus

This section describes the menus and menu items found in the Magic Cap Simulator.

File menu

Menu item	Description
`Open Copy of Package`	**Displays a dialog box that lets you navigate to a package and open it.**
`Show/Hide Development Tools`	**Shows or hides the Examine, Discipline, Testing, Log, and Reset Tests menus.** **When you hide the development tools, Magic Developer also ignores Assert and Complain macros. See "Custom debugging macros" on page 74 for more information. In addition, Magic Developer does not display "Method not found" messages when you send an invalid message to an object.**
`Exit`	**Quits the Magic Cap Simulator.**

Edit menu

Menu item	Description
Cut	If there's a text selection, cuts the selected text and places it on the Windows clipboard. If the tote bag contains a stamp, cuts the stamp and places it on the Windows clipboard.
Copy	If there's a text selection, makes a copy of the selected text on the Windows clipboard. If the tote bag contains a stamp, copies the stamp and places it on the Windows clipboard.
Paste	Pastes the contents of the Windows clipboard into Magic Cap Simulator. You can paste Windows text, images, sounds, and fonts.

Hardware menu

Menu item	Description
Com1 set to Serial/ Com1 set to Modem/ No need to toggle port	Displays the status of the Com1 port.
Enable/Disable Modem	Allows Magic Cap to use a modem plugged into your PC. After selecting this item you may need to quit and restart Magic Cap before it will start using the modem. Note: not all modems are compatible with Magic Cap.
New Card in Simulated Slot 1...	Simulates inserting a new RAM card into slot 1. The title bar of the simulator indicates that a new card is installed.
Insert Card into Simulated Slot 1...	Displays a dialog box that lets you navigate to a simulated RAM card and insert it into slot 1.
Eject Card from Simulated Slot 1	Ejects the simulated RAM card in slot 1.
New Card in Simulated Slot 2...	Simulates inserting a new RAM card into slot 2. The title bar of the simulator indicates that a new card is installed.
Insert Card in Simulated Slot 2...	Displays a dialog box that lets you navigate to a simulated RAM card and insert it into slot 2.
Eject Card from Simulated Slot 2	Ejects the simulated RAM card in slot 2.
Connect/Disconnect Simulated Phone Line	Simulates connecting a telephone line. This item is used primarily for testing the Phone line connected window.
Connect/Disconnect Simulated Hardware Keyboard	When a hardware keyboard is connected to the DataRover 840, the on-screen keyboard doesn't appear automatically when Magic Cap expects you to type text.
Start with 2 MB Additional RAM	Simulates having additional main memory. After selecting this item you must quit and restart Magic Cap.
Power On/Off	Simulates toggling device power switch.
Force Warm Reset	Simulates resetting device (without removing batteries).

Examine menu

Menu item	Description
Show/Hide Inspector	Shows or hides the object inspector.
Dump Inspector Target	Writes text description of inspected object to the log file.

Menu item	Description
`Dump Inspector Target Deep`	Writes text description of inspected object and objects referred to by its fields (except weak fields) to the log file.
`Dump Package`	Writes text description of all objects and indexicals in the current package to the log file.
`Can Move/Copy/Stretch Everything`	Allows all viewable objects to be moved, copied, and stretched regardless of their individual flag settings.

Discipline menu

Menu item	Description
`Validate Inspector Target`	Calls Validate method for the inspected object.
`Validate Package`	Calls Validate method for all objects in the current package.
`Validate System`	Calls Validate method for all system objects.
`Validate System and Active Packages`	Calls Validate method for all system and package objects.
`Faster Validates`	Tells the simulator not to run the longer, more thorough validations
`Start/Stop Leak Checking`	Enables/disables Magic Cap's storage leak detector.
`Require Card/Form Alignment`	Enables/disables a complaint in the Simulator that will fire when it detects cards and forms that are not aligned.
`Require Inherited Calls`	If this item is checked, Magic Cap executes a user break every time an overridden method executes without calling its inherited implementation.
`Require View Cache Coherency`	Checks the Magic Cap view cache for consistency. This item is used internally by Icras, Inc..

Testing menu

Menu item	Description
`Go To TestSite Scene`	Shows the Magic Cap TestSite scene. This scene and most of the items in this menu are used internally by Icras, Inc..
`Execute Standard System Test`	Performs the Magic Cap standard system test.
`Show Testing Timing Data`	Causes the Simulator to collect a list of times for the various test suites.
`Start/Stop Journaling`	Starts or stops recording a journal of Magic Cap actions which can be replayed later.
`Present/Retract Sample Announcements`	Creates or removes several announcements to test the announcement carousel.
`Simulate Device Contrast`	Simulates the grays used on personal DataRover 840 LCD screens. You should set your monitor to at least 16 grays to use this feature.
`Place Fried Egg in Name Bar`	It's a long story.
`Simulate Multiple Services`	Simulates registering for multiple electronic mail services to test features such as Collect from: window.
`Fill Memory`	Simulates filling up all memory.
`Return Memory to Normal`	Ends simulation of full memory.
`Collect Garbage`	Initiates system level garbage collection right away instead of waiting for the times that it would normally occur. Any garbage that is collected is listed in the log file. This can be useful for finding memory leaks during package development.

Menu item	Description
`Send Package`	**Displays a dialog box that lets you select a package and e-mail it.**

Log menu

Check a command on the Log menu to enable Magic Cap to write that type of information to the log file.

Menu item
`Encodings`
`Encoding Details`
`Communication Details`
`Garbage Collection`
`Redraws`
`Text Formatting`
`CRCs`
`Idles`
`Search`
`Copy Engine`
`Log Output to Debugger`
`Scheduler`
`Actors`
`Current Actors`
`Actor Timings`
`Screen Update Timings`
`Semaphores`
`Semaphore Details`
`Uncaught Exceptions`

Reset Tests menu

The commands on the Reset Tests menu simulate a DataRover reset during a critical operation, such as renumbering. These tests are typically for internal use only.

Construction tools

Building a package with Magic Developer involves spending part of your development time modifying objects within Magic Cap. Magic Cap provides two operating modes. End users work in the default **normal mode**, where the DataRover 840 behaves in the manner described by the user documentation. Developers work in **construction mode**, which gives you enhanced control over objects in the interface.

In construction mode, you can use **construction tools** to refine the appearance of objects by changing their size, placement, color, and other attributes. Construction tools help you assemble and modify the raw materials supplied by classes and static objects.

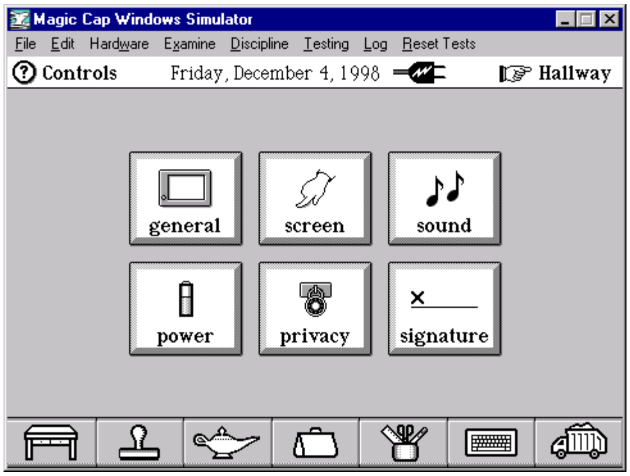
Construction mode behaves differently in a Magic Cap DataRover 840 and in the Magic Cap Simulator. You should use construction mode in the Magic Cap Simulator, because it provides additional tools for developers not found on a DataRover 840. The following sections show you how to turn on construction mode and use the Magic Cap construction tools.

Turning on construction mode

To turn on construction mode:

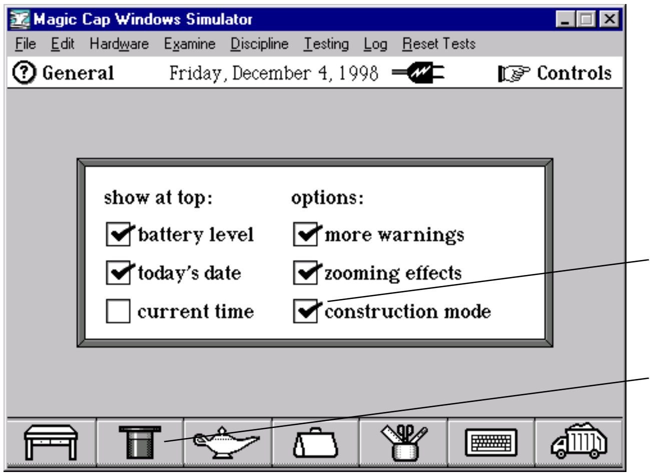
1. Go to the Hallway, find the Controls door and click it.

   A group of buttons appears.


Magic Cap Windows Simulator
File Edit Hardware Examine Discipline Testing Log Reset Tests
Controls
Friday, December 4, 1998
Hallway
general
screen
sound
power
privacy
signature

2. Click *general*.

   Two groups of check boxes appear.


Magic Cap Windows Simulator
File Edit Hardware Examine Discipline Testing Log Reset Tests
General
Friday, December 4, 1998
Controls

show at top:

☑ battery level

☑ today's date

☐ current time

options:

☑ more warnings

☑ zooming effects

☑ construction mode

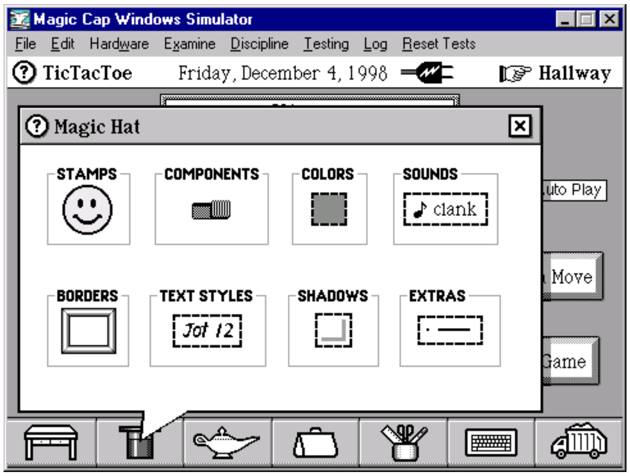
Turn on the construction mode check box...

...and the Stamper icon changes to the Magic hat

3. Turn on the check box for *construction mode*.

   The Stamper icon at the bottom of the Magic Cap screen changes to a Magic Hat icon to indicate that you are in construction mode.

To display the construction tools any time you are in construction mode, click the Magic Hat:



Some of the changes you’ll see in construction mode include the following:

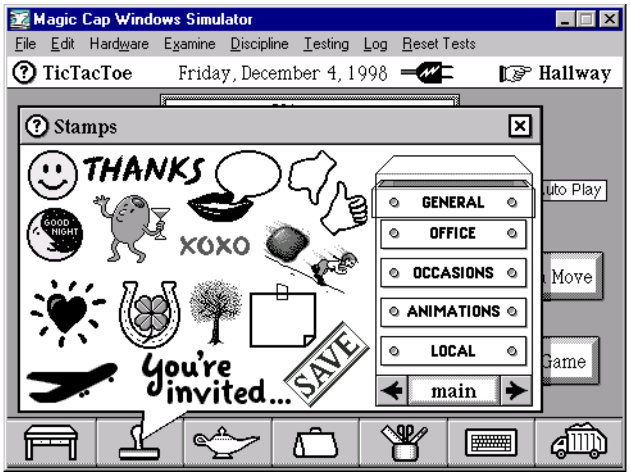
- The Magic Hat replaces the Stamper at the bottom of the screen. The Magic Hat is a combined catalog and storehouse of objects. In it you’ll find boxes, text fields, controls, and lots of coupons for modifying viewable objects on the screen. The Magic Hat window also contains the original contents of the Stamper window.
- The Tool holder at the bottom of the screen contains additional construction tools.

When you turn on construction mode in the manner described above, you are changing the mode until you either exit the simulator or return to normal mode again. Instead of turning on construction mode, you can also access the Magic Hat window temporarily to perform a single operation.

To access the Magic Hat window for a single operation:

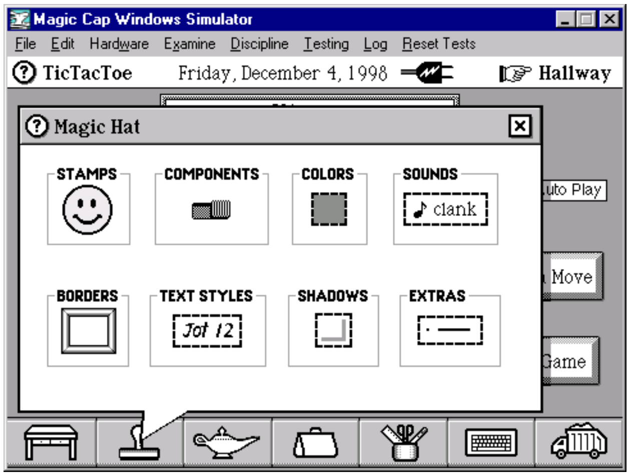
1. Click the Stamper icon to open.

   The Stamps window appears.



2. Hold down the *Control* key and click the title bar of the Stamps window.

   The Magic Hat window appears.

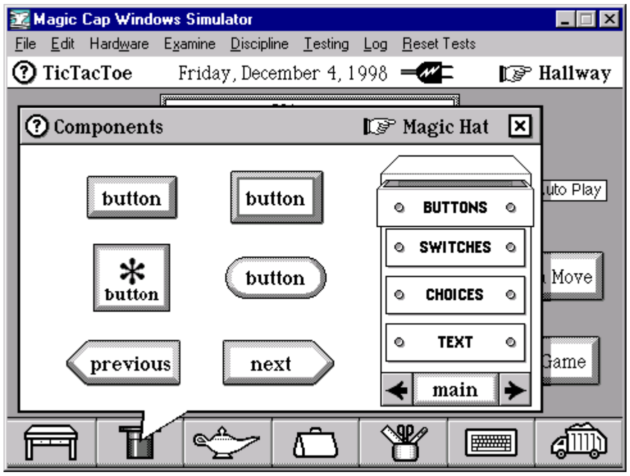


After you make a choice and use it, the Magic Hat window disappears. You remain in normal mode the entire time, and the Magic Hat icon does not replace the Stamper.

The Magic Hat window

The Magic Hat window is arranged into eight categories, as shown in the illustration in “Turning on construction mode” on page 38. If you click one of the category boxes, Magic Cap displays the tools for that category in the window. For example, clicking the Stamps category displays the contents of the original Stamper.

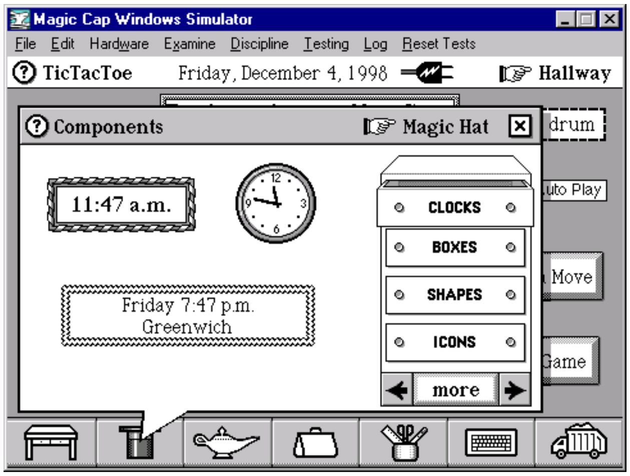
The window for some categories also displays a chest of drawers you can use to select different types of tools in that category. For example, the drawers in the Components category let you choose buttons, switches, choices, or text tools.



To dispense a stamp or other object from one of the categories, click it. The Magic Hat window closes, and a copy of the object is placed in the scene. If the object is an intangible attribute like a color, Magic Cap provides a temporary container called a **coupon** to hold it until you apply it to something.

To dispense more than one object from a category, hold down the *Control* key and click an object. The object falls out of the Magic Hat into the scene behind the window, and the window remains open. If you let go of the *Control* key and click another item, the object is deposited in the scene and the Magic Hat window closes.

When there are too many objects to fit in a chest of drawers, a choice box at the bottom of the chest lets you choose from additional sets of drawers. For example, if you select the *more* choice in the Components window, you'll see the following drawers:



Coupons

Coupons hold intangible attributes from the Magic Hat, such as colors and borders, that you can apply to viewable objects. When you drag the coupon to an object that accepts it, the object displays that attribute.

To use a coupon:

1. Click the Magic Hat icon.

   The Magic Hat window appears.

2. Click any of the coupon categories: Colors, Sounds, Borders, Text Styles, Shadows, or Extras.

   The Magic Hat window displays the items in that category.

3. Do either of the following:

   - Click an item.
   - Hold down the *Control* key to select multiple items, then release the *Control* key and click the final item.

   The Magic Hat window closes and each item appears in the scene as a coupon with a thick, dashed border around it.

4. Drag each coupon to an object that accepts it, then release it. If an object can accept a coupon, the object changes color when you drag to it.

   The object displays the attribute.

If objects have more than one part, you can drop coupons onto separate parts to perform different functions. For example, boxes have separate content and border parts. You can drop different color coupons into the border and content parts to set them to different colors.

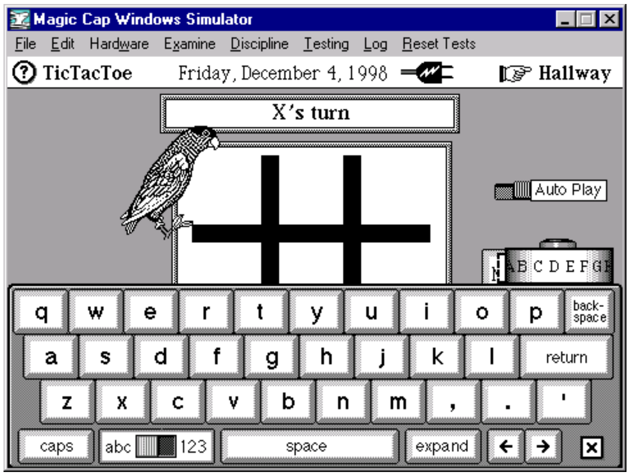
Text coupons

Use a text coupon to create a label for objects such as buttons, telecards, notebook pages, and name cards.

To make a text coupon:

1. Hold down the *Control* key and click the Keyboard icon at the bottom of the screen.

   The Keyboard appears with a label maker above it.


Magic Cap Windows Simulator
File Edit Hardware Examine Discipline Testing Log Reset Tests
TicTacToe
Friday, December 4, 1998
Hallway
X's turn
Auto Play
A B C D E F G
q w e r t y u i o p back-space
a s d f g h j k l return
z x c v b n m , . '
caps abc 123 space expand

2. Type the text for the label.
3. Drag the coupon away from the label maker.

   The label maker disappears, and a text coupon remains in the scene.

4. Drag the text coupon to an object that accepts it, then release it. If an object can accept a text coupon, it changes color when you drag to it.

   The object displays the text as a label.

Stamps

As you look through Magic Cap and Magic Cap packages, you'll see cards decorated with all sorts of small pictures: company logos, postage stamps, beasts, cacti, stylized skyscrapers. These pictures are called **stamps**. You can add them to any card and position them anywhere you like.

The Stamper contains a collection of different stamps that you can use as clip art in the cards of your packages. The contents of the Stamper drawers are summarized in the following table.

Drawer	Function
general	The general drawer contains a diverse selection of decorative stamps, an animation, the sound-holding lips stamp, and a sticky note.
office	The stamps in the office drawer are designed for marking office messages and commercial transactions.
occasions	The occasions drawer holds decorative stamps for assorted holidays and special events.
animations	The animations drawer provides a selection of animated characters that you can use to decorate your cards. If you tinker an animation with the Tinker tool described later in this chapter, you can control how fast it moves horizontally and vertically and how rapidly the animation's frames change. You can also determine whether the animation should turn when it hits the sides of its container.
local	Scenes can have their own stamps. If they do, these stamps will appear in the local drawer and the name of the local stamp will match that of the scene. Hallway is an example of a scene with its own stamps. Scenes in packages can also have their own stamps.
faces	The faces in the faces drawer are often useful when you're sending a message to someone and want to emphasize your mood visually.
symbols	The symbols drawer contains common symbols you might find useful for any message and for labeling pictures or maps you draw.
leisure	The leisure drawer includes stamps representing popular diversions and pastimes.
songs	Each song stamp contains a different tune. Place one of these stamps on a card, then touch the stamp to play.
misc.	This drawer contains miscellaneous stamps.

Components

The components category contains a group of user-interface tools. These are the main ingredients for building the parts of a package that people interact with. These component objects have been developed to solve a variety of user-interface problems while maintaining a consistent set of user expectations within the Magic Cap environment.

Buttons

A button is an object that a user can touch to invoke some action in a package. Buttons have two states: normal and highlighted. The highlighted state is temporary and only lasts while the button is being touched. After that, the button returns to its normal state.

Switches

A switch is a control with two positions. Change a switch's position by clicking it. You can change the sound it makes when it's touched by dropping a sound coupon on it. Switches can be collected into a group with the `ChooseOneBox` class. In this case, they behave like a group of radio buttons.

Choices

Choices are controls that let users select from a group of options. Magic Cap provides the following types of choices:

- **Choice boxes** let users select an option from a list of options with text names.
- **Sliders** let users control continuously adjustable levels.
- **Meters** let users select numeric values.

Text

Text components display text entered from the keyboard. Text has many characteristics that you can set, including typeface, size, style, and alignment. You can set the characteristics of a text field with the coupons in the text styles category box.

Text components can contain many other kinds of objects, including shapes, stamps, animations, other text fields, and even scribbles from the pen. Objects placed within a text component are clipped to the containing text component. When you create a text field by pulling it out of the Magic Hat, its ability to contain other objects is turned off. To turn it on, tinker the text field with the Tinker tool and flip the *can contain* switch to the on position.

Magic Cap provides three special text components named phone, time, and numbers. These components are simple forms processing controls. For example, the numbers component accepts numeric input only.

Clocks

Clocks tell what time it is based on the time maintained in the DataRover 840. They come in several different shapes and sizes. Some clocks are analog, others are digital, and many are both. Some clocks can display the time zone as well as the time.

Boxes

Boxes organize groups of other objects. They have a border and are filled with a particular color. They're different from text fields in that boxes are primarily organizational and are designed to hold heterogeneous groups of graphical objects, not text.

Shapes

Shapes are irregular objects that share some properties of boxes—they can contain other objects, and they have a border and a color. When they are stretched with the stretch tool, they maintain their shape while being scaled horizontally or vertically.

Icons

The icons drawer contains a group of icons used in Magic Cap. If you change an icon by dropping an image coupon on a system icon, you can drop a new icon from the icons drawer to replace it.

You can create also create an image coupon from any bitmap to replace any icon.

To create an image coupon:

1. Create or open an image in a paint application.
2. Copy the image to the clipboard.
3. Start or activate the Magic Cap Simulator.
4. Paste the image into the simulator in either of the following ways:

    - To paste a two-bit image, choose Edit▶Paste.
    - To paste a one-bit image, hold down the *Shift* key and choose Edit▶Paste.

    The image appears as an image coupon in Magic Cap. If the image was in color, it is automatically converted to a grayscale.

5. Drag the image coupon to an object that accepts it, then release it. If an object can accept an image coupon, it changes color when you drag to it.

    The object displays the custom image.

Colors

You can change the color that fills an object by dropping a color coupon into it. Most viewable objects accept color coupons.

The color coupon category of the Magic Hat also contains a color mixer. If you want to experiment with a variety of colors easily, drag the color mixer from the Magic Hat window into a scene where you want to work. Drag coupons from the color mixer to objects experiment with different color combinations. When you are finished with the color mixer, drag it to the trash truck to remove it.

Sounds

A sound coupon specifies the sound an object plays when it is clicked. You can drop sound coupons into most viewable objects. If you don't specify a sound, switches and buttons will play the "touch" sound when clicked.

Drawer	Description
standard	The standard drawer contains the sounds used throughout Magic Cap to keep the user aware of what is happening. You can drop these coupons into an object to change the sound that it makes when activated.
instruments	The instruments drawer contains coupons representing all the synthesized musical instruments that are part of Magic Cap. These are the instruments used for making MIDI songs.
songs	The songs in the songs drawer use the instruments described above to make synthesized music.
phone	A Magic Cap DataRover 840 can make all the DTMF sounds that a phone needs to make.
more	The more drawer contains extra sounds, including the ever popular "no sound" coupon.

Borders

Use a border coupon to change the style of an object's border. Use the *no border* coupon to remove the border from an object. Some objects that accept borders are boxes, fields, the Inspector, and meters.

Drawer	Description
basic	The basic border drawer includes the no border coupon, which you drop on an object to rid it of an existing border.
lines	These are the simplest borders. They draw quickly and are widely used in Magic Cap.
fancy	These fancy frames might be used to set off an austere minimalist sketch.
objects	The book and clock frames in the objects drawer are useful for books and clocks.
misc.	The misc. drawer contains extra borders.

Text styles

Text style coupons contain a combination of type faces, point sizes, weights, and orientations. All text style coupons can be dropped into text fields. You can also change the appearance of an object's label by dropping a text style coupon into it.

The following table summarizes the text styles available in Magic Cap:

Style	Description
Plain	Contains 8, 10, and 12 point Sign, and 10 and 12 point Book.
Big	Contains 14 and 18 point Sign, and 14 and 18 point Book.
Styled	Contains different sizes of Book and Sign with bold, italic, and underlined styles.
Fancy	Contains the Fat Caps, Typewriter, and Jot faces.

Shadows

Many objects display shadows to improve their appearance. You can use shadow coupons to change a shadow's appearance, or to remove a shadow. You can drop shadows into most viewable objects.

If you want to experiment with a variety of shadow types easily, drag the shadow style choice box from the Magic Hat window into a scene where you want to work. Use the arrows on the choice box to select a shadow style, then drag the selected coupon out of the center section. When you are finished with the choice box, drag it to the trash truck to remove it.

Extras

The extras drawer contains many different coupons that are useful for constructing and adjusting the user interface of a package. The following table summarizes the coupons that are available:

Coupon	Options	Description
`line styles`		Drop a line style coupon into an object to change the appearance of the border that's drawn around it.
`properties`	make moveable make unmovable make copyable make uncopyable make deletable make undeletable	Many objects that you see can be modified with the view coupons you can find by choosing properties in the Magic Hat. You can use these coupons to change the behavior of objects.
`shape types`		Shape type coupons let you specify various styles for the shape of an object, such as a rectangle, a circle, a diamond, or an arrow. Shape type coupons can only be dropped into shape objects.
`misc.`	hide show contents rotate left rotate right flip vertical flip horizontal bring to front send to back put in form pull from form	The misc. drawer lets you control the geometric properties of a viewable object. Shapes, stamps, and animations can be rotated in 90-degree increments. You can drop orientation coupons into these objects to rotate or flip them. The 'put in form' coupon and the 'pull from form' coupons work with the form objects in cards.

Most of these coupons are available in the Magic Cap Simulator only. Magic Cap itself contains only Line Styles within the Magic Hat.

Tool holder

Tools are modes that you can turn on and off. When a tool is on, objects do something special when you touch them. Most of the time, you only need to touch objects to make them work. But in construction mode, you can copy, stretch, and manipulate objects in various ways.

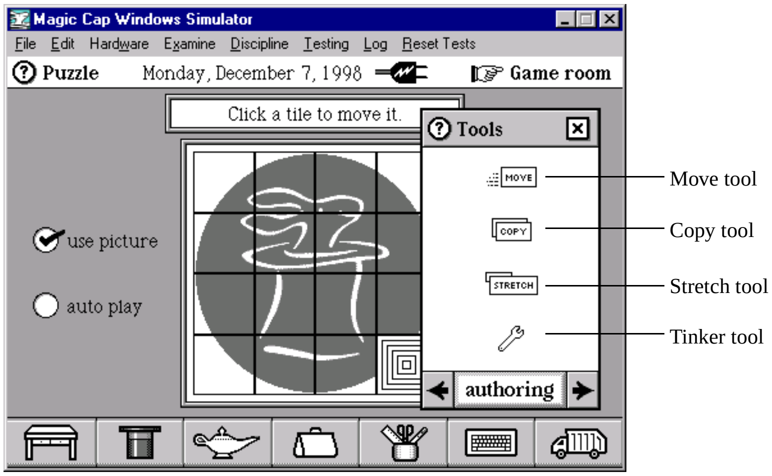
In construction mode, the Tool holder contains two extra items that let you change the layout of objects in a scene or access the Inspector. These tools only appear in the version of construction mode available in the Magic Cap Simulator.

To make the Tool holder appear, click its icon at the bottom of the screen. The Tool holder window appears and displays a set of pencils. At the bottom of the Tools window is a choice box that lets you choose among the sets of tools. In construction mode, you'll be able to use the authoring and debugging tools.

When you select a tool, the Tools window disappears and the Tool holder icon at the bottom of the screen changes to a tool indicator icon. This icon represents the current tool that will be applied to the next object you click. If an object cannot accept a tool, Magic Cap plays a sound and lets you choose another object. If you don't need a tool you have chosen, click the tool indicator icon to dismiss it.

Authoring

The authoring tools let you specify the layout of objects in a scene. You can move, copy, and stretch objects; in addition, you can change the attributes of an object that determine whether it can be moved, copied, or stretched. The authoring tools are shown in the following illustration:



Tinker tool

The tinker tool lets you examine an object and adjust its properties. The properties determine whether you can use other tools or the objects in the Magic Hat to modify the object's appearance and behavior.

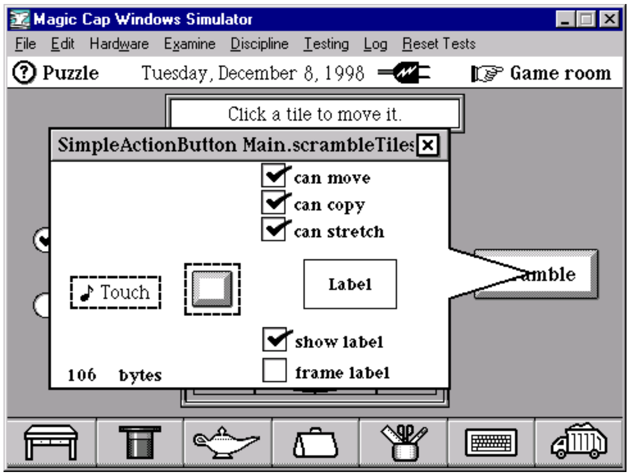
For example, many objects contain properties such as *can move*, *can copy*, and *can stretch*. These properties determine whether you can use the move, copy, and stretch tools to modify the object's appearance. Similarly, the tinker window of an object may display a frame and a sound, indicating that you can use frame and sound coupons to modify the object.

To use the tinker tool:

1. Click the *tool holder* and choose *authoring*.
2. Click the *tinker* tool.

   The tinker tool icon replaces the tool holder.
3. Click the object you want to examine.

   The tinker window for the object appears, and the tool holder icon replaces the tinker tool in the button bar.



SimpleActionButton Main.scrambleTile

☑ can move

☑ can copy

☑ can stretch

☑ show label

☐ frame label

106 bytes

Check boxes in the tinker window control many of an object's properties. If an object can have a label, the label chooser lets you select one of 15 positions for the label by clicking the position you want. A data string in the bottom left corner of the window shows how many bytes of memory the object uses.

The settings that are shown when you open an object will vary for different classes of objects. If you see coupons that you like, you can drag them out of the open window and drop them on other objects. However, you can't drop coupons into the tinker window to change an object. Just drop the coupons on the object itself to make the changes.

You can move the tinker window around by grabbing its title bar, and you can click the close box to close the window.

Move tool

You can move some objects by simply sliding them, without using the move tool. However, most objects require you to use the move tool to change their position. It's always worth trying an object to see if you can move it without the move tool. If you can't, try the move tool.

To move an object with the move tool:

1. Click the *tool holder* and choose *authoring*.
2. Click the *move* tool.

   The move tool icon replaces the tool holder.
3. Slide the object around on the screen.

If you can't move an object with the move tool, you might need to turn on its *can move* property. Tinker the object as described in "Tinker tool" on page 49, turn on its *can move* property, then use the move tool.

When you move an object on the screen, it's drawn with a shadow to indicate that it's not attached to the scene.

Copy tool

You can use the copy tool to create a copy of an object. To make a copy, click the copy tool, then slide out a new copy of an object. The original object will remain in its old location while you slide the newly created copy.

You may have to set the *can copy* property as described in "Tinker tool" on page 49 before using the copy tool.

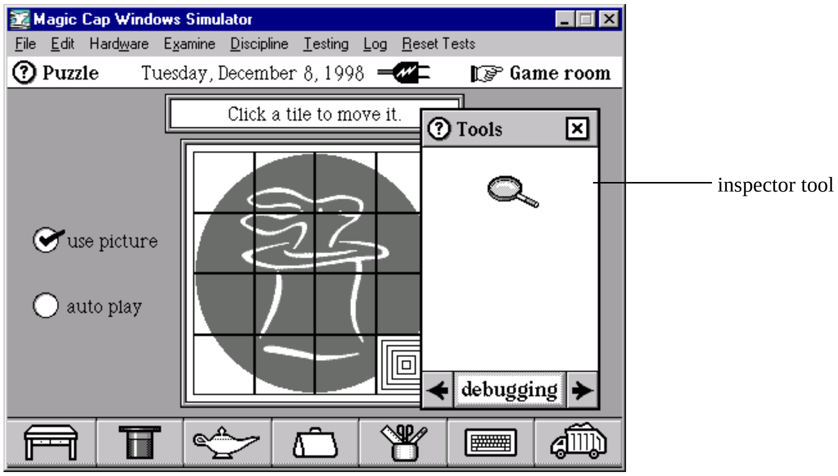
Stretch tool

You can use the stretch tool to resize most objects. To do this, click the stretch tool, then drag an object to its new size.

You may have to set the *can stretch* property as described in "Tinker tool" on page 49 before using the stretch tool.

Debugging with the Inspector

The debugging tool added to the tool holder in construction mode lets you access the Inspector. The Inspector provides detailed information about objects in Magic Cap.


Magic Cap Windows Simulator
File Edit Hardware Examine Discipline Testing Log Reset Tests
Puzzle
Tuesday, December 8, 1998
Game room
Click a tile to move it.
use picture
auto play
Tools
debugging
inspector tool

There are several ways to access the Inspector: through the Tool holder, through menus in the Magic Cap Simulator, and through the Magic Developer toolbar in Microsoft Visual C++. Here's how to access the Inspector from the Tool holder:

1. Click the tool holder and choose *debugging*.
2. Click the *inspector* tool.

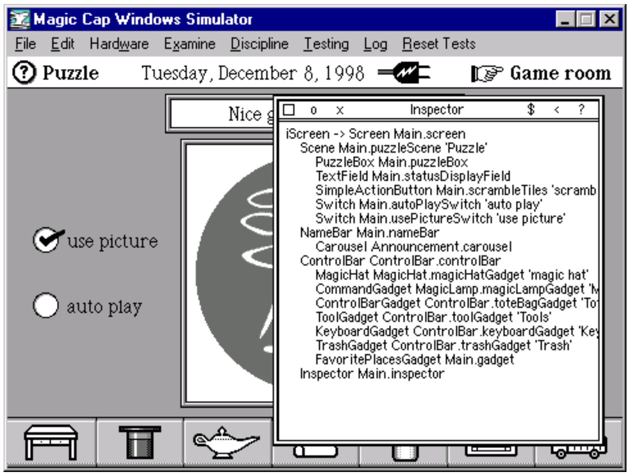
   The inspector tool icon replaces the tool holder.
3. Click the object you want to inspect.

   The inspector window for the object appears, and the tool holder icon replaces the inspector tool in the button bar.

For more information on using the Inspector, see “Inspector” on page 52.

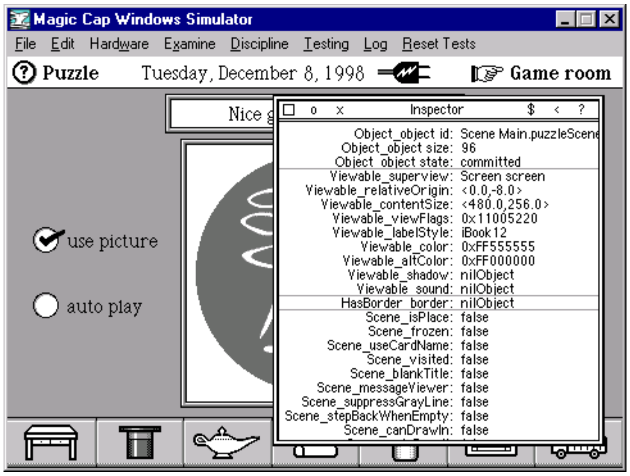
Inspector

The Inspector is a tool for interactively analyzing Magic Cap objects. With it you will point at objects and look at their contents. The Examine menu in the Magic Cap Simulator has several items that work with the Inspector.


Magic Cap Windows Simulator
File Edit Hardware Examine Discipline Testing Log Reset Tests
Puzzle
Tuesday, December 8, 1998
Game room
Nice g
use picture
auto play
□ o x Inspector $ < ?

```
iScreen -> Screen Main.screen
  Scene Main.puzzleScene 'Puzzle'
    PuzzleBox Main.puzzleBox
    TextField Main.statusDisplayField
    SimpleActionButton Main.scrambleTiles 'scramb
    Switch Main.autoPlaySwitch 'auto play'
    Switch Main.usePictureSwitch 'use picture'
  NameBar Main.nameBar
    Carousel Announcement.carousel
  ControlBar ControlBar.controlBar
    MagicHat MagicHat.magicHatGadget 'magic hat'
    CommandGadget MagicLamp.magicLampGadget 'M
    ControlBarGadget ControlBar.toteBagGadget 'To
    ToolGadget ControlBar.toolGadget 'Tools'
    KeyboardGadget ControlBar.keyboardGadget 'Key
    TrashGadget ControlBar.trashGadget 'Trash'
    FavoritePlacesGadget Main.gadget
  Inspector Main.inspector
```

If the inspector is started from the Examine menu or from the Magic Developer toolbar in Microsoft Visual C++, it displays a list of the current hierarchy of viewable objects, called the view list (see the above illustration). To see information about any object in the view list, click the item in the inspector; the display changes, as shown in the illustration below.


Magic Cap Windows Simulator
File Edit Hardware Examine Discipline Testing Log Reset Tests
Puzzle
Tuesday, December 8, 1998
Game room
Nice g
use picture
auto play
□ o x Inspector $ < ?

```
         Object_object id:  Scene Main.puzzleScene
       Object_object size:  96
      Object_object state:  committed
      Viewable_superview:  Screen screen
 Viewable_relativeOrigin:  <0.0,-8.0>
    Viewable_contentSize:  <480.0,256.0>
      Viewable_viewFlags:  0x11005220
     Viewable_labelStyle:  iBook12
          Viewable_color:  0xFF555555
       Viewable_altColor:  0xFF000000
         Viewable_shadow:  nilObject
          Viewable_sound:  nilObject
        HasBorder_border:  nilObject
           Scene_isPlace:  false
            Scene_frozen:  false
       Scene_useCardName:  false
           Scene_visited:  false
        Scene_blankTitle:  false
     Scene_messageViewer:  false
  Scene_suppressGrayLine:  false
Scene_stepBackWhenEmpty:  false
         Scene_canDrawIn:  false
```

The Inspector displays values for each of the fields of the object. The object reference, object size, and object state are displayed at the top of the Inspector, above the field information. A gray line separates this information from the field information.

The name of the class that defines each field is prefixed to the field name. A gray line separates the field information in one class from the field information in another class. If you hold down the *Control* key and click the $ in the title bar, the class name disappears.

If the information you're looking at doesn't fit in the Inspector window, you can scroll it by sliding the text, or you can use the stretch tool to enlarge the Inspector's window. You can also stretch the Inspector without getting the stretch tool by simply dragging on its frame, an operation that requires precise pointing.

If you click a field that holds an object reference, the Inspector will change to show you information about that object.

The Inspector provides five controls in its title bar that let you use additional features: *O*, *X*, <, *$*, and *?*. If you click *?* in the title bar, then click an object, the Inspector displays that object's fields. This will change the Inspector's target object to the newly selected object. Click < to make the Inspector display the fields of the object that it displayed just previously. Click the *X* to switch between seeing the view list and an object's fields.

You can use the Inspector to begin tinkering with any object in the view list, including objects that you can't touch, like the screen object. To do this, click the *O* (for Open) in the Inspector's title bar while it's displaying information about the object that you want to inspect.

Fields that show an object reference can be displayed in one of two modes. The first, shown above, describes each object by showing its class name, the object's name, if it has one, and the instance definition tag. This information is useful for high-level debugging and general exploration. Clicking the *$* in the title bar changes the display to show the object reference for each object. Object references are especially useful for low-level debugging. Click the *$* again to switch between displaying instance definition tags and object references.

You can move the Inspector around on the screen by holding down the *Control* key while you press on it anywhere and drag it, or by dragging the Inspector's title bar, with or without the *Control* key. You can drop coupons into the Inspector to change its appearance, including its border, text style, color, and so forth.

To inspect a system object, including the system root list, hold down the *Control* key and click the *X* in the title bar. To inspect any class object, inspect any object of that class, then click the object reference.

When you're done with the Inspector, you can get rid of it by clicking its close box in the top-left corner.

Dumping single objects

As you create your packages, moving back and forth between Microsoft Visual C++ and the Magic Cap Simulator, you may sometimes want to convert just a few objects to text rather than dumping all objects to text, as described in “Modifying viewable objects with construction tools” on page 28. While you’re running the Magic Cap Simulator, you can convert any object to its text representation by displaying the object in the Inspector, then choosing Examine▶Dump Inspector Target.

When you dump an object this way, the text representation is written to a file called `Log`, located in the same directory as the workspace file (.dsw file) for your project. You can dump several objects by repeatedly aiming the Inspector, then choosing Examine▶Dump Inspector Target. You can then open the file in Microsoft Visual C++ to see the dumped objects.

You can also choose Examine▶Dump Inspector Target Deep to dump the inspected object and all objects that it’s related to, such as subviews, targets, object lists, and other objects referred to by fields. This feature, called **deep dumping**, is useful if you want to take an object and the objects that support it and move them to another package. Deep dumping dumps the same objects that would be copied if the object were copied; that is, it doesn’t dump any fields marked `weak` in the class definition file.

4

Bowser Jo

Bowser Jo is an HTML-based class browser for Magic Cap classes. With it you can look at reference information for a class and its associated methods and fields. You can also enter a search string to find matching classes, methods and fields. Since Bowser Jo is based on HTML and Java, you access it in your web browser.

Launching Bowser Jo

To launch Bowser Jo:

1. Launch your web browser.
2. Open the file `index.html` in the `docs\htmlhelp` subdirectory of the directory in which Magic Developer is installed:

   a. In your web browser, choose File▶Open.

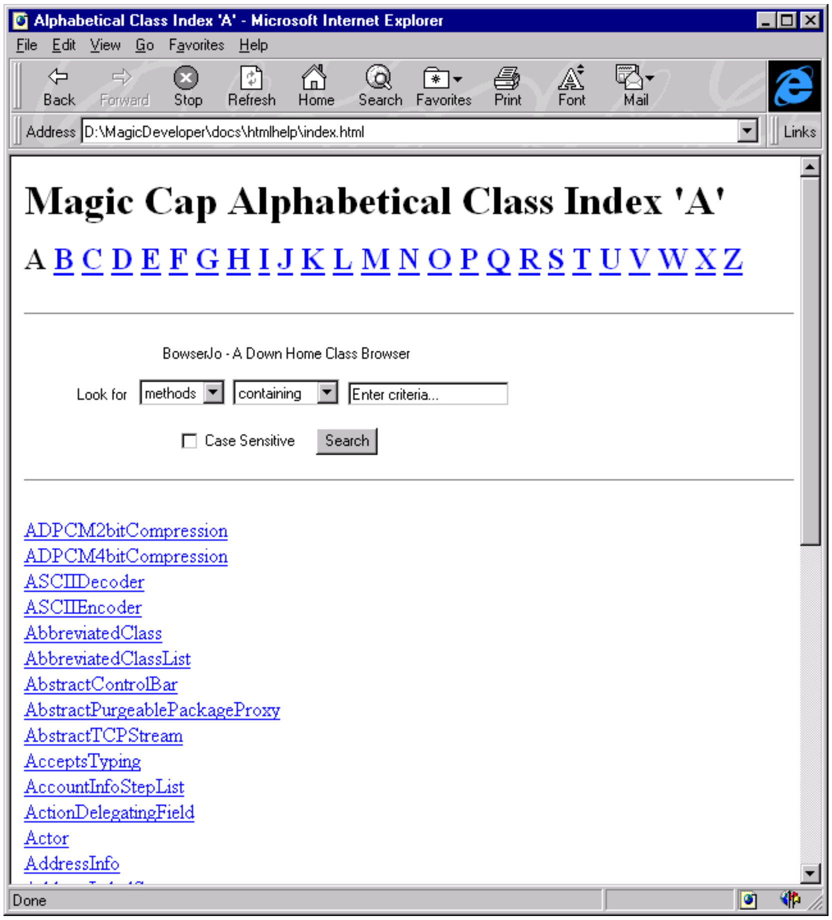
   A dialog box lets you enter a file name.

   b. Click Browse.

   A dialog box lets you navigate to a file.

   c. Navigate to `index.html`, then choose Open and OK to close the two dialog boxes.

The Bowser Jo applet appears in your web browser.



Alphabetical Class Index 'A' - Microsoft Internet Explorer

Address D:\MagicDeveloper\docs\htmlhelp\index.html

Magic Cap Alphabetical Class Index 'A'

A B C D E F G H I J K L M N O P Q R S T U V W X Z

BowserJo - A Down Home Class Browser

Look for methods containing Enter criteria...

□ Case Sensitive Search

ADPCM2bitCompression
ADPCM4bitCompression
ASCIIDecoder
ASCIIEncoder
AbbreviatedClass
AbbreviatedClassList
AbstractControlBar
AbstractPurgeablePackageProxy
AbstractTCPStream
AcceptsTyping
AccountInfoStepList
ActionDelegatingField
Actor
AddressInfo

Done

3. Optionally set a bookmark for Bowser Jo to launch it again easily.

Using Bowser Jo

There are two ways of searching for information with Bowser Jo: alphabetically and by string matching.

Searching alphabetically

The row at the top of the Bowser Jo window provides alphabetical hyperlinks for navigating through the list of Magic Cap constructs. The list at the bottom of the window corresponds to the letter of the alphabet selected at the top.

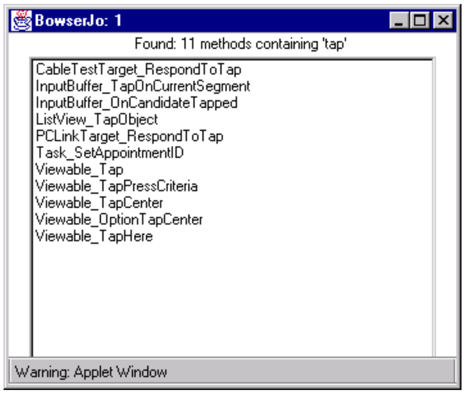
Searching by string matching

Sometimes you'll want to get a list of Magic Cap constructs that are associated with a string. You can search for methods, fields, or classes that contain a specific string.

For example, to find all the methods with the string "tap" in their name:

1. Select *methods* in the first drop-down list.
2. Select *containing* in the second drop-down list.
3. Type the string **tap** in the *Enter criteria* field.
4. Click Search.

   Bowser Jo returns a list of methods meeting the criteria you specified.



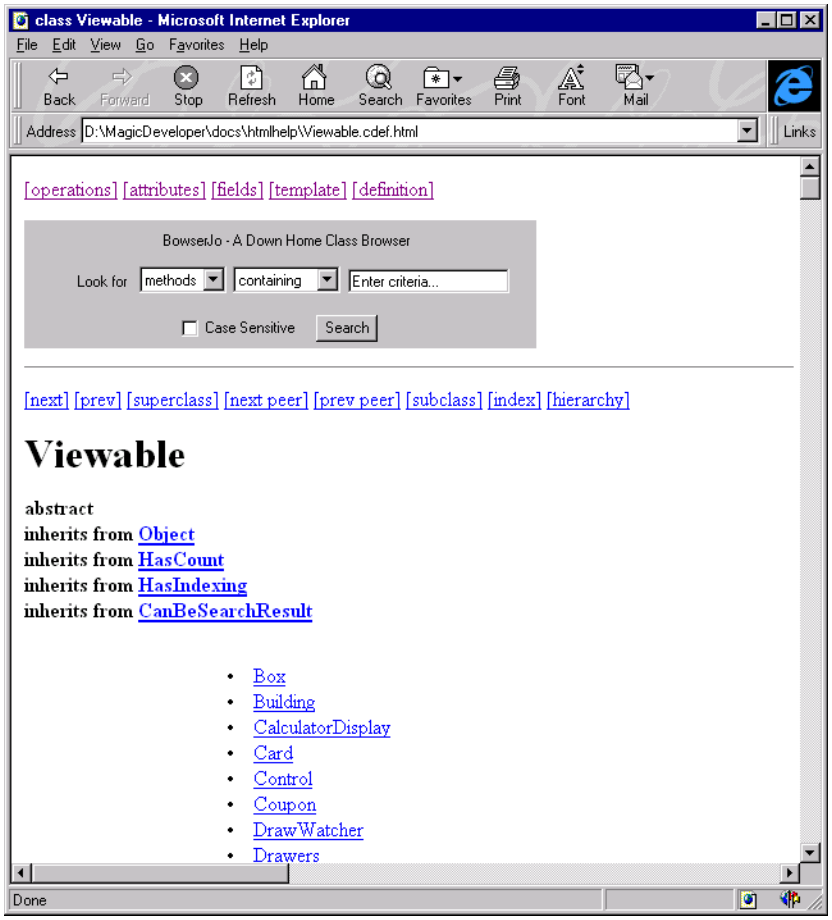
Found: 11 methods containing 'tap'

```
CableTestTarget_RespondToTap
InputBuffer_TapOnCurrentSegment
InputBuffer_OnCandidateTapped
ListView_TapObject
PCLinkTarget_RespondToTap
Task_SetAppointmentID
Viewable_Tap
Viewable_TapPressCriteria
Viewable_TapCenter
Viewable_OptionTapCenter
Viewable_TapHere
```

Warning: Applet Window

5. Double-click one of the methods in this list

Bowser Jo displays complete information about the method.



class Viewable - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites Print Font Mail
Address D:\MagicDeveloper\docs\htmlhelp\Viewable.cdef.html
Links

[operations] [attributes] [fields] [template] [definition]

BowserJo - A Down Home Class Browser

Look for methods containing Enter criteria...

□ Case Sensitive Search

[next] [prev] [superclass] [next peer] [prev peer] [subclass] [index] [hierarchy]

Viewable

abstract
inherits from Object
inherits from HasCount
inherits from HasIndexing
inherits from CanBeSearchResult

- Box
- Building
- CalculatorDisplay
- Card
- Control
- Coupon
- DrawWatcher
- Drawers

5

Debugging Tools

You can use two environments to debug Magic Cap packages:

- Microsoft Visual C++ and the Magic Cap Simulator
- The GNU source-level debugger (GDB) and the DataRover 840

Magic Developer extends both Microsoft Visual C++ and GDB with custom commands to provide access to Magic Cap features.

To begin debugging, build your package targeting the Simulator, then use the Microsoft Visual C++ debugging tools to stabilize your package. After you have a working project, build the package targeting a DataRover 840, then download the package to the DataRover 840 and use GDB to finish debugging.

Debugging in Microsoft Visual C++

Magic Developer provides the following tools to extend the debugging capabilities of Microsoft Visual C++:

- Commands on the toolbar that let you examine objects and their memory usage.

  These commands are summarized in “Summary of Magic Developer toolbar commands” on page 60.

- Macros to insert directly in your source code that write messages to the log file and break into the debugger.

  These macros are described in “Custom debugging macros” on page 74.

The rest of this section shows you how to use the Magic Developer enhancements to Microsoft Visual C++. For general information about debugging in Microsoft Visual C++, see the Microsoft documentation.

Creating a debug build

Before you can begin debugging in Microsoft Visual C++, you must create a debug build of your package that targets the Magic Cap Simulator.

1. Launch Microsoft Visual C++.
2. Create a debug build of your package that targets the Magic Cap Simulator.

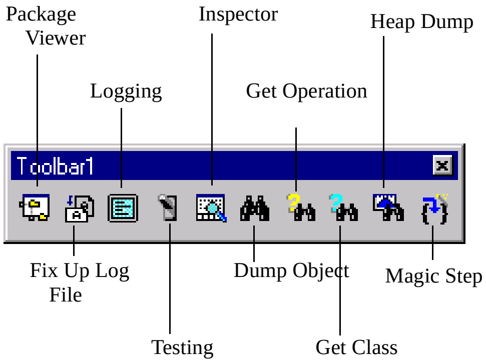
   See “How to build a package” on page 17 for complete information.
3. If the Microsoft Visual C++ Workspace Viewer is open, close it.

   You cannot open files in a Magic Developer project from the Microsoft Visual C++ Workspace Viewer; instead, you use the Magic Developer Package Viewer. See “Displaying and managing the contents of a package” on page 61.

Summary of Magic Developer toolbar commands

When you enable the Magic Developer debugging capabilities within Microsoft Visual C++, you create the toolbar that provides access to the Magic Developer extensions. See “Enabling Magic Developer” on page 17 of *Magic Developer 3.2 Installation*.

The following tools are available on the Magic Developer toolbar:



The following table summarizes these toolbar commands:

Tool	Description
`Package Viewer`	**Displays the organization of a package and lets you manage the files within it. See "Displaying and managing the contents of a package" on page 61.**
`Fix Up Log File`	**Opens a copy of the log file. See "Opening a copy of the log file" on page 64.**
`Logging`	**Lets you specify the type of information which Magic Cap writes to the log file. See "Setting log file preferences" on page 65.**
`Testing`	**Runs Simulator tests from within Magic Developer. See "Running Simulator tests" on page 66.**
`Inspector`	**Lets you interactively analyze Magic Cap objects. See "Inspector" on page 52.**
`Dump Object`	**Displays the field values of an object or indexical when you are debugging. See "Examining objects and indexicals" on page 67.**
`Get Operation`	**Displays information about an operation. See "Viewing operation information" on page 69.**
`Get Class`	**Displays information about a class. See "Viewing class information" on page 69.**
`Heap Dump`	**Examines clusters or the heap when you are debugging. See "Examining memory usage" on page 70.**
`Magic Step`	**Lets you step into a dispatched method and execute it one line at a time. See "Stepping into a dispatch call" on page 73.**

Displaying and managing the contents of a package

The Package Viewer replaces the Microsoft Visual C++ Workspace Viewer; it provides a convenient way to display the organization of a package and to manage the files in a package.

Displaying package organization

The Package Viewer provides three views of the current package in Microsoft Visual C++:

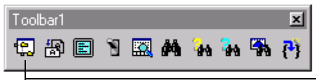
- The list of source code files that create the package
- The list of classes in the package
- The list of objects in the package

You can use the Package Viewer any time you are working in Microsoft Visual C++; however, the lists of classes and objects are available only after you have built your project.

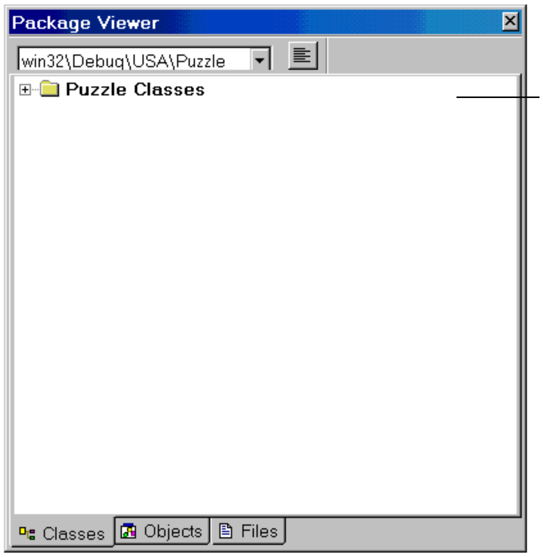
To display the organization of a package with the Package Viewer:

1. Click the Package Viewer button on the Magic Developer toolbar.

   The Magic Developer Package Viewer opens.



When you click the Package Viewer icon...

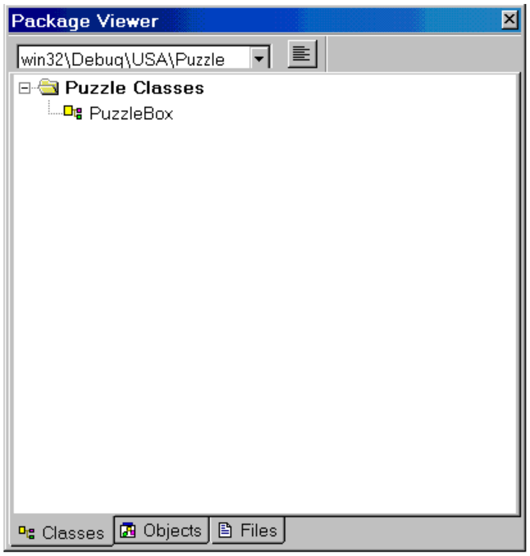


...Magic Developer displays the Package Viewer

2. Click either the Classes, Objects, or Files tab to specify the package view you want to examine.

3. If the folder in the left pane is closed, double-click it or click the *plus sign*.

   The folder opens and displays the list of classes, objects, or files.



Managing project files

In addition to letting you view the organization of a package, the Package Viewer lets you perform the following project-management tasks:

- Insert a file
- Remove a file
- Create a new makefile
- Open a file for editing

Inserting a file

To insert a file in a project:

1. Open the project workspace and display the Files panel of the Package Viewer.
2. Right-click the Package Viewer and choose Insert.

   A dialog box lets you select the file to insert in the project.
3. Double-click the file, or select it and choose Open.

   The file is inserted into the project and the makefile.

Removing a file

To remove a file from a project:

1. Open the project workspace and display the Files panel of the Package Viewer.
2. Right-click the file you want to remove and choose Remove.

   Magic Developer removes the file from the Files list in the Package Viewer and from the makefile. The file remains on your hard disk.

Creating a new makefile

To create a new makefile for a project:

1. Open the project workspace and display the Files panel of the Package Viewer.
2. Right-click the Package Viewer and choose Recreate Makefile.

   Magic Developer searches the current directory and creates a makefile out of all relevant files.
3. Close the project workspace and then re-open it.

Opening a file

To open a project file:

1. Open the project workspace and display the Files panel of the Package Viewer.
2. Double-click the file you want to open.

   Magic Developer opens the file in Microsoft Visual C++.

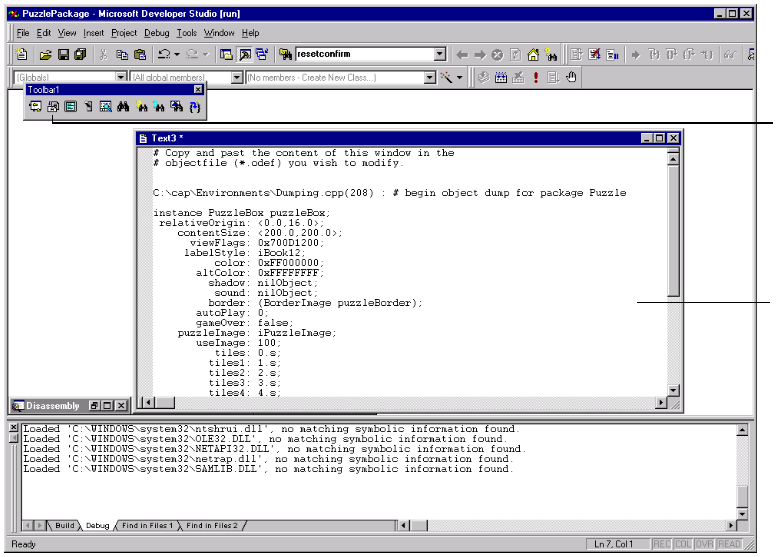
Opening a copy of the log file

You can open a copy of the log file quickly by using the `FixUpLogFile` macro which is available on the Magic Developer toolbar. When you use this toolbar command, Magic Developer creates a copy of the log file in memory and opens it in a text window in Microsoft Visual C++. You can then modify the object definitions in the file, cut or copy the objects to other source files, and save the text file without touching the current log file.

For example, suppose you want to paste the source code for an object you have interactively modified in the Simulator into the `Objects.odef` file.

1. Use the tools in the Simulator to modify the object. See “Construction tools” on page 37.
2. Open the Inspector in the Simulator, then dump the object to the log file. See “Dumping single objects” on page 54.
3. In Microsoft Visual C++, click the FixUpLogFile icon on the Magic Developer toolbar.

   Magic Developer opens a new window which contains a copy of the log file.



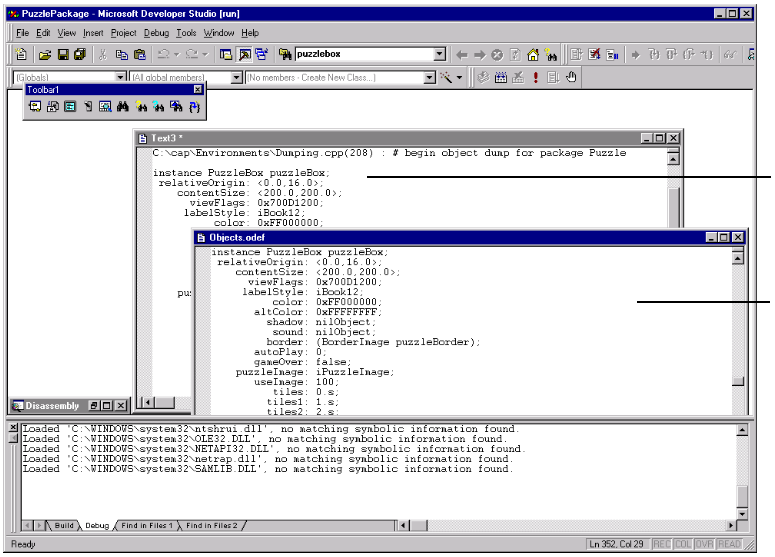
```
# Copy and past the content of this window in the
# objectfile (*.odef) you wish to modify.


C:\cap\Environments\Dumping.cpp(208) : # begin object dump for package Puzzle

instance PuzzleBox puzzleBox;
 relativeOrigin: <0.0,16.0>;
    contentSize: <200.0,200.0>;
      viewFlags: 0x700D1200;
     labelStyle: iBook12;
          color: 0xFF000000;
       altColor: 0xFFFFFFFF;
         shadow: nilObject;
          sound: nilObject;
         border: (BorderImage puzzleBorder);
       autoPlay: 0;
       gameOver: false;
    puzzleImage: iPuzzleImage;
       useImage: 100;
          tiles: 0.s;
         tiles1: 1.s;
         tiles2: 2.s;
         tiles3: 3.s;
         tiles4: 4.s;
```

```
Loaded 'C:\WINDOWS\system32\ntshrui.dll', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\OLE32.DLL', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\NETAPI32.DLL', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\netrap.dll', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\SAMLIB.DLL', no matching symbolic information found.
```

When you click the FixUpLogFile icon...

...Magic Developer displays the contents of the log file in a new window

4. In Microsoft Visual C++, copy the dumped object to the clipboard.
5. In Microsoft Visual C++, open the `Objects.odef` file, then replace the object definition with the copy on the clipboard and save the file.



```
C:\cap\Environments\Dumping.cpp(208) : # begin object dump for package Puzzle

instance PuzzleBox puzzleBox;
 relativeOrigin: <0.0,16.0>;
    contentSize: <200.0,200.0>;
      viewFlags: 0x700D1200;
     labelStyle: iBook12;
          color: 0xFF000000;
```

```
instance PuzzleBox puzzleBox;
 relativeOrigin: <0.0,16.0>;
    contentSize: <200.0,200.0>;
      viewFlags: 0x700D1200;
     labelStyle: iBook12;
          color: 0xFF000000;
       altColor: 0xFFFFFFFF;
         shadow: nilObject;
          sound: nilObject;
         border: (BorderImage puzzleBorder);
       autoPlay: 0;
       gameOver: false;
    puzzleImage: iPuzzleImage;
       useImage: 100;
          tiles: 0.s;
         tiles1: 1.s;
         tiles2: 2.s;
```

```
Loaded 'C:\WINDOWS\system32\ntshrui.dll', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\OLE32.DLL', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\NETAPI32.DLL', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\netrap.dll', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\SAMLIB.DLL', no matching symbolic information found.
```

Copy the object definition to the clipboard...

...then paste it over the existing object definition in Objects.odef

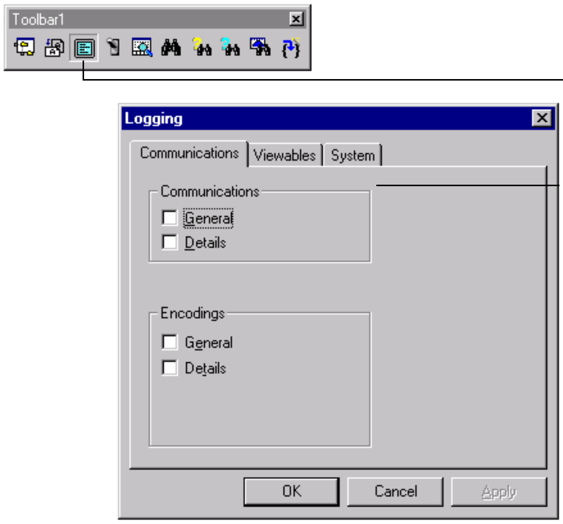
Setting log file preferences

You can use the Logging dialog box to specify the type of information which Magic Cap writes to the log file. The preferences available in the Logging dialog box are the same as the choices available from the Log menu in the Magic Cap Simulator. See “Log menu” on page 37 for more information.

To specify preferences for the log file:

1. Click the Logging button on the Magic Developer toolbar.

   The Magic Developer Logging dialog box opens.



Communications
□ General
□ Details

Encodings
□ General
□ Details

When you click the Logging icon...

...Magic Developer displays the Logging dialog box

2. Check the items which you want Magic Cap to write to the log file in each panel of the Logging dialog box, then click OK.

Running Simulator tests

You can use the Testing button on the Magic Developer toolbar to run Simulator tests from within Microsoft Visual C++. When you click the Testing button, Magic Developer displays the three panels of the Testing dialog box:

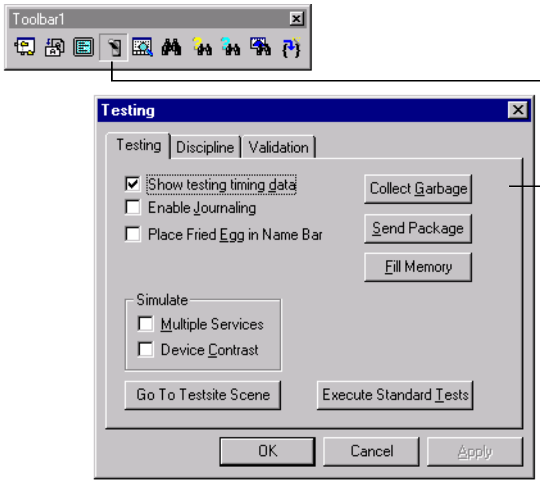
Note: The choices available on the Testing panel are the same as the choices available from the Testing menu in the Magic Cap Simulator. See “Testing menu” on page 35 for more information.

- The choices available on the Validation and Discipline panels are the same as the choices available from the Discipline menu in the Magic Cap Simulator. See “Discipline menu” on page 35 for more information.

To run Simulator tests from Microsoft Visual C++:

1. Click the Testing button on the Magic Developer toolbar.

The Magic Developer Testing dialog box opens.



Toolbar1

Testing

Testing | Discipline | Validation

☑ Show testing timing data
☐ Enable Journaling
☐ Place Fried Egg in Name Bar

Simulate
☐ Multiple Services
☐ Device Contrast

When you click the Testing icon...

...Magic Developer displays the Testing dialog box

2. Do either of the following:

- Check the tests which you want to perform, then click OK.

  The Simulator performs the tests.

- Click the button which represents the test you want to perform.

  The Simulator performs the test immediately.

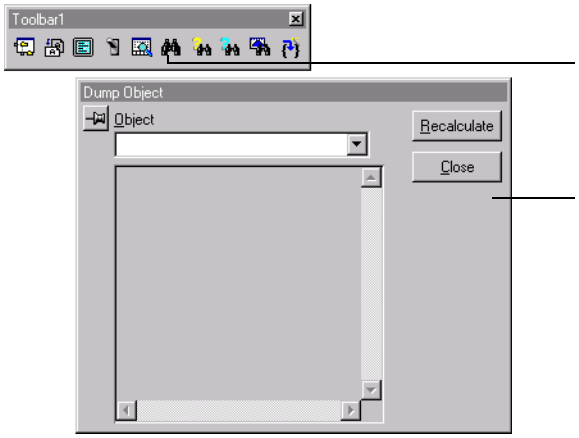
Examining objects and indexicals

You can use the Dump Object window to examine the field values of an object or indexical when you are debugging. You may have to set a breakpoint so the debugger breaks in an appropriate context to examine certain objects.

To use the Dump Object window:

1. Click the Dump Object icon on the Magic Developer toolbar.

   Magic Developer displays the Dump Object window.

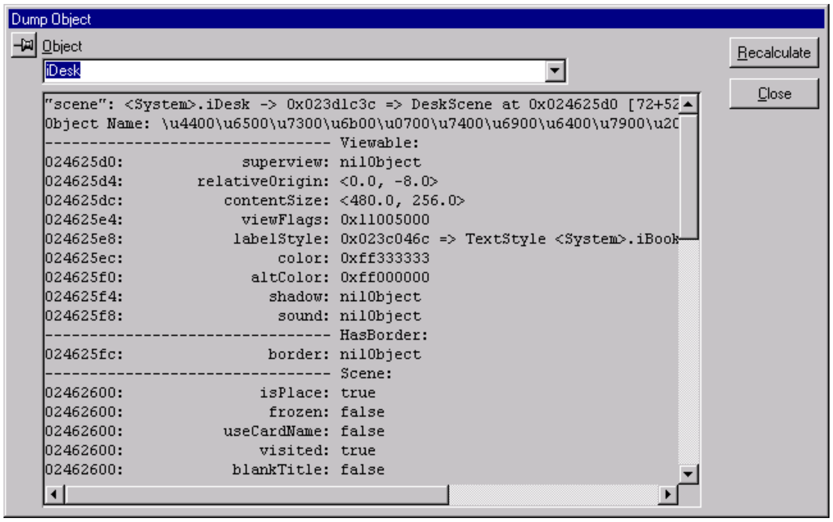


When you click the Dump Object icon...

...Magic Developer displays the Dump Object window

2. Type an expression to evaluate, such as an indexical name or an object tag, in the Object field, then press *Enter*.

   Magic Developer dumps the object for you to examine.



```
"scene": <System>.iDesk -> 0x023d1c3c => DeskScene at 0x024625d0 [72+52
Object Name: \u4400\u6500\u7300\u6b00\u0700\u7400\u6900\u6400\u7900\u2C
-------------------------------- Viewable:
024625d0:             superview: nilObject
024625d4:        relativeOrigin: <0.0, -8.0>
024625dc:           contentSize: <480.0, 256.0>
024625e4:             viewFlags: 0x11005000
024625e8:            labelStyle: 0x023c046c => TextStyle <System>.iBook
024625ec:                 color: 0xff333333
024625f0:              altColor: 0xff000000
024625f4:                shadow: nilObject
024625f8:                 sound: nilObject
-------------------------------- HasBorder:
024625fc:                border: nilObject
-------------------------------- Scene:
02462600:               isPlace: true
02462600:                frozen: false
02462600:           useCardName: false
02462600:               visited: true
02462600:            blankTitle: false
```

3. To examine an object referenced by the object you are viewing, double-click the new object’s hexadecimal object reference to the right of the colon (:).

   Magic Developer launches another instance of the window and dumps the new object.

The information in the Dump Object window is similar to the information displayed in the Magic Cap Simulator's Inspector. If the object has a name, it is displayed above the first dotted line in the Dump Object window, along with the package to which the object belongs and its hexadecimal address.

The name of the class that defines each field is displayed to the right of the dotted lines. These dotted lines separate the field information in one class from the field information in another class.

Each field either has a scalar value or else points to another object or indexical. If the referenced object or indexical has a name, it is displayed with the hexadecimal value. In the above illustration, for example, the value of the `border` field in the `HasBorder` class is `nilObject`. However, the `labelStyle` field of `Viewable` points to the `TextStyle` object.

You can modify the value of a field by clicking the value. Type a new value for the field and press *Enter* to change it. Be careful to change field values (especially those for ROM objects) in valid ways. Changing field values improperly can result in crashes or unpredictable results.

About the object reference

The object reference is an identifier that Magic Cap assigns to an object dynamically at run time. There are three common ways to determine an object's reference:

- Set a break point in one of the object's methods and look at the `self` parameter.
- Use the Dump Object tool in Microsoft Visual C++ to dump the object.
- While looking at the contents of another object, you may get an object's reference from the value of a field.

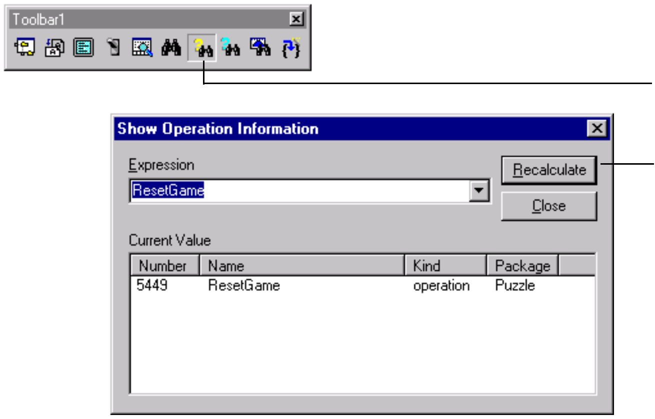
Viewing operation information

If you know either the operation number or name, you can view other information about the operation by using the Magic Developer toolbar.

To view operation information:

1. Click the Get Operation button on the Magic Developer toolbar.

   The Show Operation Information dialog box opens.



Show Operation Information

Expression: ResetGame

Current Value

Number	Name	Kind	Package
5449	ResetGame	operation	Puzzle

When you click the Get Operation icon...

...Magic Developer displays the Show Operation Information dialog box

2. Enter either the operation name or number in the Expression field, then click Recalculate.

   The number, name, kind, and package are displayed in the Current Value panel.

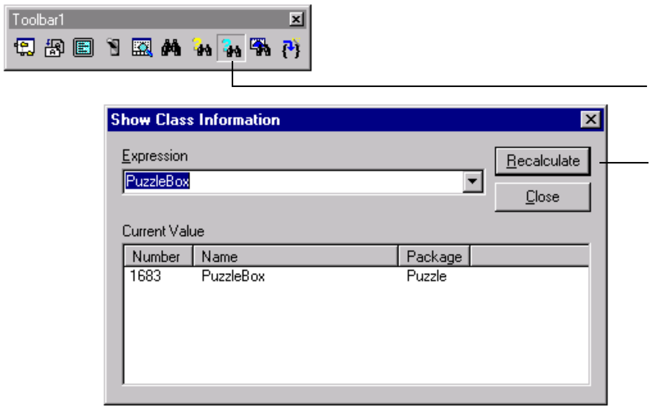
Viewing class information

If you know either the class number or name, you can view other information about the class by using the Magic Developer toolbar.

To view class information:

1. Click the Get Class button on the Magic Developer toolbar.

   The Show Class Information dialog box opens.



Show Class Information

Expression: PuzzleBox

Current Value

Number	Name	Package
1683	PuzzleBox	Puzzle

When you click the Get Class icon...

...Magic Developer displays the Show Class Information dialog box

2. Enter either the class name or number in the Expression field, then click Recalculate.

   The number, name, and package are displayed in the Current Value panel.

Examining memory usage

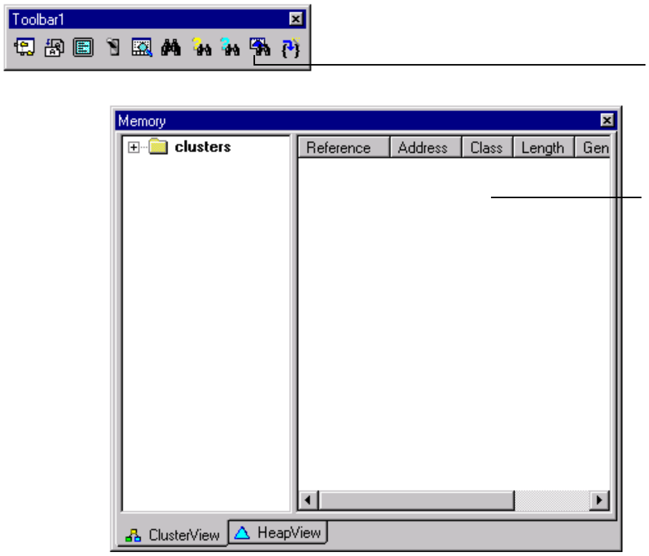
You can use the Memory window to examine clusters or the heap when you are debugging.

- A **cluster** is a collection of objects that belong to the same package.
- The **heap** is a relocatable block of memory that contains objects.

To use the Memory window:

1. Click the Heap Dump icon on the Magic Developer toolbar.

   Magic Developer displays the Memory window.



When you click the Heap Dump icon...

...Magic Developer displays the Memory window

Examining memory clusters

The ClusterView panel of the Memory window lets you examine memory clusters and the objects they contain.

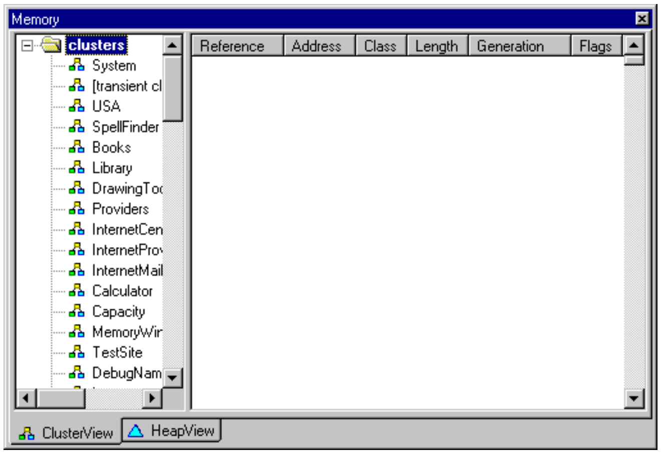
To examine a cluster:

1. Click the ClusterView tab in the Memory window.

   Magic Developer displays the ClusterView panel.

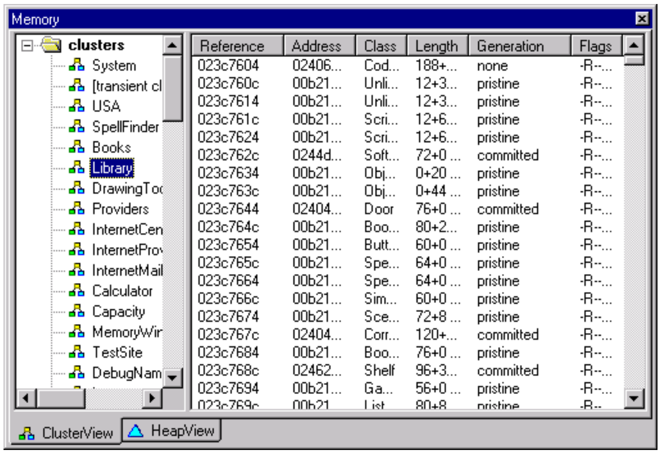
2. If the *clusters* folder in the left pane is closed, double-click it or click the *plus sign*.

   The folder opens and displays the clusters currently loaded.



3. Double-click the cluster you want to examine.

   The objects in the cluster are displayed in the right pane, sorted by their object references. You can adjust the width of a column in this pane by dragging the edge of the column label.



Reference	Address	Class	Length	Generation	Flags
023c7604	02406...	Cod...	188+...	none	-R--...
023c760c	00b21...	Unli...	12+3...	pristine	-R--...
023c7614	00b21...	Unli...	12+3...	pristine	-R--...
023c761c	00b21...	Scri...	12+6...	pristine	-R--...
023c7624	00b21...	Scri...	12+6...	pristine	-R--...
023c762c	0244d...	Soft...	72+0 ...	committed	-R--...
023c7634	00b21...	Obj...	0+20 ...	pristine	-R--...
023c763c	00b21...	Obj...	0+44 ...	pristine	-R--...
023c7644	02404...	Door	76+0 ...	committed	-R--...
023c764c	00b21...	Boo...	80+2...	pristine	-R--...
023c7654	00b21...	Butt...	60+0 ...	pristine	-R--...
023c765c	00b21...	Spe...	64+0 ...	pristine	-R--...
023c7664	00b21...	Spe...	64+0 ...	pristine	-R--...
023c766c	00b21...	Sim...	60+0 ...	pristine	-R--...
023c7674	00b21...	Sce...	72+8 ...	pristine	-R--...
023c767c	02404...	Corr...	120+...	committed	-R--...
023c7684	00b21...	Boo...	76+0 ...	pristine	-R--...
023c768c	02462...	Shelf	96+3...	committed	-R--...
023c7694	00b21...	Ga...	56+0 ...	pristine	-R--...
023c769c	00b21	List	80+8	pristine	-R--

4. Optionally double-click an object reference to examine the details of an object.

   Magic Developer launches the Dump Object window and displays the object's details.

As you continue debugging, you can refresh the list of clusters at any time by right-clicking the left pane of the ClusterView panel and choosing Refresh.

Examining heaps and non-relocatable memory

The HeapView panel of the Memory window lets you examine heaps (relocatable blocks of memory) as well as non-relocatable blocks of memory while you are debugging.

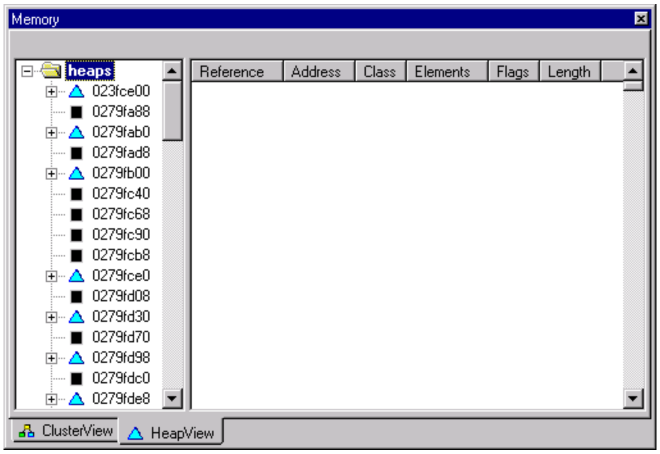
To examine a heap:

1. Click the HeapView tab in the Memory window.

   Magic Developer displays the HeapView panel.

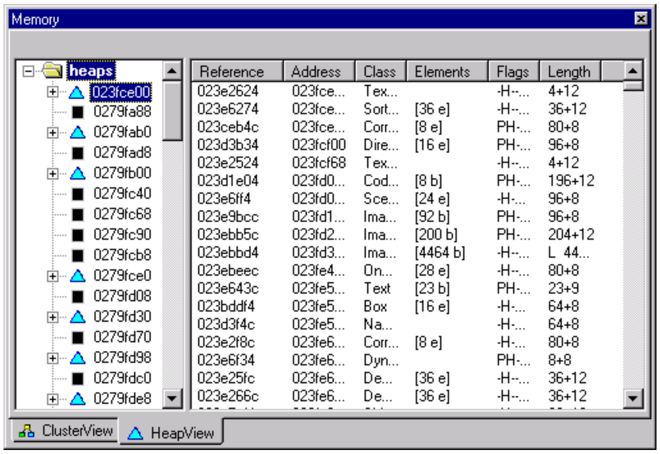
2. If the *heaps* folder in the left pane is closed, double-click it or click the *plus sign*.

   The folder opens and displays the heaps.



3. Double-click the heap you want to examine.

   The objects in the heap are displayed in the right pane, sorted by their object references. You can adjust the width of a column in this pane by dragging the edge of the column label.



Reference	Address	Class	Elements	Flags	Length
023e2624	023fce...	Tex...		-H--...	4+12
023e6274	023fce...	Sort...	[36 e]	-H--...	36+12
023ceb4c	023fce...	Corr...	[8 e]	PH-...	80+8
023d3b34	023fcf00	Dire...	[16 e]	PH-...	96+8
023e2524	023fcf68	Tex...		-H--...	4+12
023d1e04	023fd0...	Cod...	[8 b]	PH-...	196+12
023e6ff4	023fd0...	Sce...	[24 e]	-H-...	96+8
023e9bcc	023fd1...	Ima...	[92 b]	PH-...	96+8
023ebb5c	023fd2...	Ima...	[200 b]	PH-...	204+12
023ebbd4	023fd3...	Ima...	[4464 b]	-H--...	L 44...
023ebeec	023fe4...	On...	[28 e]	-H--...	80+8
023e643c	023fe5...	Text	[23 b]	PH-...	23+9
023bddf4	023fe5...	Box	[16 e]	-H-...	64+8
023d3f4c	023fe5...	Na...		-H-...	64+8
023e2f8c	023fe6...	Corr...	[8 e]	-H-...	80+8
023e6f34	023fe6...	Dyn...		PH-...	8+8
023e25fc	023fe6...	De...	[36 e]	-H--...	36+12
023e266c	023fe6...	De...	[36 e]	-H--...	36+12

4. Optionally double-click an object reference to examine the details of an object.

   Magic Developer launches the Dump Object window and displays the object's details.

Each heap can contain an unlimited number of nested heaps. To improve performance, Magic Developer does not initially analyze a heap; instead, it displays the heap in the HeapView panel with a plus sign. When you double-click a heap, Magic Developer analyzes it and either displays a list of nested heaps or removes the plus sign.

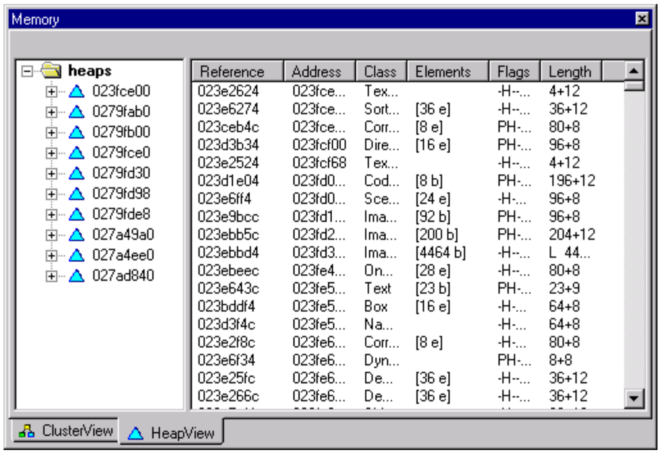
As you continue debugging, you can refresh the list of heaps and non-relocatable blocks of memory at any time by right-clicking the left pane of the HeapView panel and choosing Refresh.

Non-relocatable blocks of memory usually hold C++ data structures that are allocated with the new statement. Since these structures may contain pointers instead of references, the memory cannot be moved.

To suppress the list of non-relocatable blocks of memory

- Right-click the left pane of the HeapView panel and deselect *Show non-relocatable blocks.*

  Magic Developer removes the non-relocatable blocks from the HeapView panel.



heaps
- 023fce00
- 0279fab0
- 0279fb00
- 0279fce0
- 0279fd30
- 0279fd98
- 0279fde8
- 027a49a0
- 027a4ee0
- 027ad840

Reference	Address	Class	Elements	Flags	Length
023e2624	023fce...	Tex...		-H--...	4+12
023e6274	023fce...	Sort...	[36 e]	-H--...	36+12
023ceb4c	023fce...	Corr...	[8 e]	PH-...	80+8
023d3b34	023fcf00	Dire...	[16 e]	PH-...	96+8
023e2524	023fcf68	Tex...		-H--...	4+12
023d1e04	023fd0...	Cod...	[8 b]	PH-...	196+12
023e6ff4	023fd0...	Sce...	[24 e]	-H-...	96+8
023e9bcc	023fd1...	Ima...	[92 b]	PH-...	96+8
023ebb5c	023fd2...	Ima...	[200 b]	PH-...	204+12
023ebbd4	023fd3...	Ima...	[4464 b]	-H--...	L 44...
023ebeec	023fe4...	On...	[28 e]	-H--...	80+8
023e643c	023fe5...	Text	[23 b]	PH-...	23+9
023bddf4	023fe5...	Box	[16 e]	-H-...	64+8
023d3f4c	023fe5...	Na...		-H-...	64+8
023e2f8c	023fe6...	Corr...	[8 e]	-H-...	80+8
023e6f34	023fe6...	Dyn...		PH-...	8+8
023e25fc	023fe6...	De...	[36 e]	-H--...	36+12
023e266c	023fe6...	De...	[36 e]	-H--...	36+12

Stepping into a dispatch call

When a called subroutine is a function, you can use the Step Into command (*F11*) in the Microsoft Visual C++ debugger to execute it one line at a time. When a called subroutine is a dispatched method, however, the Microsoft Visual C++ debugger steps over it instead of stepping into it.

A **dispatched method** is a subroutine that is called through the Magic Cap object dispatcher. At runtime, the object dispatcher determines which method to call; that information cannot be determined when the package is compiled.

The Microsoft Visual C++ debugger does not recognize the assembly language which the object dispatcher is written in. Use the MagicStep command (*Shift+F8*) on the Magic Developer toolbar to step into a dispatched method.

For example, the file `puzzle.cpp` in the `samples\Puzzle` subdirectory of the directory in which Magic Developer is installed contains the following call to the dispatched method `ResetGame`:

```
Method void
PuzzleBox_ResetConfirm(Reference self)
{
  if (OptionKey()) {
    ResetGame(self);
    return;
  }

  PerformWithConfirmation(iConfirmationWindow,
            iScrambleTilesPrompt,
            self, operation_ResetGame,
            iMagicSound,
            iPuzzleLogo,
            true);
}
```

If you place a breakpoint at the `ResetConfirm` method and then use MagicStep to execute each line in it, the debugger drops into the `ResetGame` method so you can execute each line in `ResetGame` individually. However, if you use the Microsoft Visual C++ Step Into command to execute each line in `ResetConfirm`, `ResetGame` will execute as a single block of code.

MagicStep steps only into a dispatched method if full symbolic information is available for it; that is, if a dispatched method is either within your own package or another package for which you have symbolic information, you can use MagicStep to step into it. If Magic Developer cannot find symbolic information or if the called subroutine is not a dispatch, MagicStep behaves like the Microsoft Visual C++ Step Into command.

Custom debugging macros

Magic Developer provides custom macros to assist you in debugging packages that you create. The following table summarizes the debugging macros:

Macro	Behavior
`Assert(`*`condition`*`)`	**If the condition is false, breaks into the debugger, writes a message to the message window, and writes a message to the log file.**
`Complain((`*`message`*`))`	**Breaks into the debugger, writes the message to the message window, and writes the message to the log file.**
`DebugMessage((`*`message`*`))`	**Writes the message text to the message window and to the log file.**
`Log((`*`message`*`))`	**Writes the message text to the log file.**

Because these macros require a debug version of Magic Cap, they work only with the Magic Cap Simulator; they are ignored when you are debugging with GDB on a DataRover 840.

The `message` strings in these macros can include any `sprintf` parameter list. For example:

```
Log(("the number %d is too small", number));
```

If you hide the development tools menus in the Simulator, Magic Developer ignores the following macros:

- Assert macros
- Complain macros

See “File menu” on page 33 for more information.

Breaking into the debugger

When you are testing a package in the Magic Cap Simulator, the `Complain` and `Assert` macros interrupt program execution and break into the debugger.

An **assertion** lets you verify run-time conditions in your code. You can use an assertion to make sure that a particular object is not nil, see if a value is within a specified range, check if one value is less than another, or perform any other test.

The `Assert` macro uses a condition to perform the test that you specify. If the condition is false, the Simulator breaks into the debugger, displays a message in the Microsoft Visual C++ message window, and writes a message to the log file.

Here are some examples of assertions:

```
/*
 * Make sure that a sound is not nil before fooling around with it.
 */
Assert(Sound(self) != nilObject);

/*
 * Make sure that an object is a member of a particular class.
 */
Assert(Implements(oldObject, Coupon_));

/*
 * Make sure that a value is in range.
 */
Assert(spoonCount < 1000);

/*
 * See if a cached value is still good.
 */
Assert(EqualBox(border, &cached));
```

Do not confuse the `Assert` macro with the `Complain` macro. The `Assert` macro only breaks into the debugger when its condition is false; the `Complain` macro breaks into the debugger whenever it executes. `Complain` also displays a message in the Microsoft Visual C++ message window and writes the message into the log file.

Writing messages to the log file

If you don’t want to break into the debugger, you can use the `Log` or `DebugMessage` macros to notify you when certain code is executing. Both `Log` and `DebugMessage` write a message that you specify to the log file; in addition, `DebugMessage` displays the message in the Microsoft Visual C++ message window.

Debugging in GDB

The GNU source-level debugger (GDB) is a freeware debugger that you can use to debug C++ applications. Magic Developer extends the debugging capabilities of GDB so you can use it to debug Magic Cap packages.

GDB has two important benefits:

- GDB is a *source-level* debugger that maintains a relationship between the software's source code and what is being executed in the Magic Cap run-time environment. The most beneficial example of this parallelism is that a developer can step through the software's source code and examine data structures in the syntax of the programming language used to develop the software.
- GDB is a *remote* debugger. GDB runs on your PC while the software you are testing runs on a DataRover 840. A serial cable connects the two systems for downloading software, sending commands, and retrieving data.

Magic Developer extends GDB to provide special features related to Magic Cap software development, including tools for inspecting and dumping objects and memory clusters. These features are summarized in "The Magic Developer extensions to GDB" on page 78.

The rest of this section shows you how to use the Magic Developer extensions to GDB. For general information about debugging in GDB, see the book *Debugging with GDB* by Richard Stallman and Roland Pesch in either of the following locations:

- In the file `gdb.pdf` in the `docs\sdkdocs` subdirectory of the directory in which Magic Developer is installed.
- At `http://www.eecs.tulane.edu/www/htdocs/gdb/gdb_toc.html` on the World Wide Web.

Launching GDB

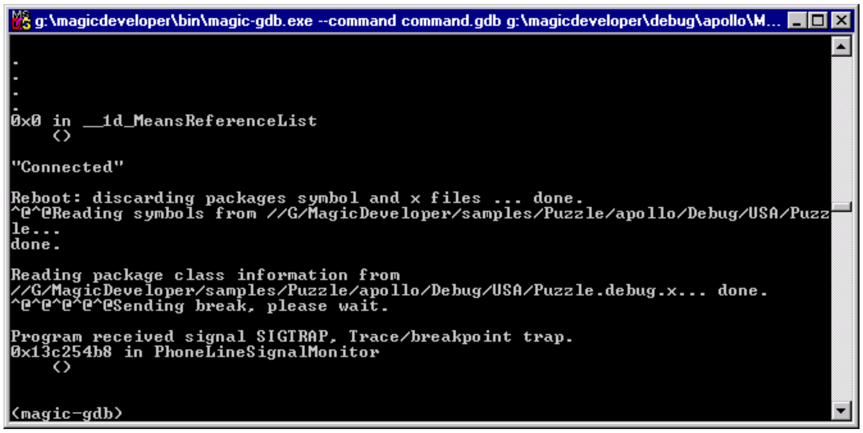
1. Create a release build of the project, targeting the appropriate DataRover 840, as described in "How to build a package" on page 17.
2. Download the build to the DataRover 840 as described in "Downloading a package to a storage card" on page 30.
3. Enter Monitor mode by cold booting while holding down the *Option* button. See "How to perform a cold boot" on page 30.

   The screen remains blank; however, the device is now on and in Monitor mode.
4. Make sure that the cable connects the serial port of your PC to the Magic Bus port of the DataRover 840.
5. On the PC, open a command prompt window, change to the directory that contains your package, then run the batch file `debug.bat`.

   GDB begins running, boots the DataRover 840, and loads the package.

6. On the PC, press *Ctrl+C* to interrupt program execution on the DataRover 840.

   The PC screen displays the debugging prompt, (*magic-gdb*).


g:\magicdeveloper\bin\magic-gdb.exe --command command.gdb g:\magicdeveloper\debug\apollo\M...

```
.
.
.
.
0x0 in __1d_MeansReferenceList
    ()

"Connected"

Reboot: discarding packages symbol and x files ... done.
^@^@Reading symbols from //G/MagicDeveloper/samples/Puzzle/apollo/Debug/USA/Puzz
le...
done.

Reading package class information from
//G/MagicDeveloper/samples/Puzzle/apollo/Debug/USA/Puzzle.debug.x... done.
^@^@^@^@^@Sending break, please wait.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x13c254b8 in PhoneLineSignalMonitor
    ()


(magic-gdb)
```

Examining objects with GDB

You typically use GDB to track program execution and inspect data structures. For example, the `MakeAMove()` method of the sample puzzle project executes every time a user attempts to move a tile in the sliding puzzle game. Suppose you want to examine the object referenced by the `MakeAMove()` method:

1. Launch the puzzle game, then interrupt program execution as described in “Launching GDB” on page 76.

2. At the GDB command prompt, set the breakpoint by entering the command `break MakeAMove`.

   The screen displays the following:

   ```
   (magic-gdb) break MakeAMove
   Breakpoint 1 at 0x3c9400: file Puzzle.cpp, line 487.
   ```

3. At the GDB command prompt, restart program execution by entering the command `cont`.

   The screen displays the following:

   ```
   (magic-gdb) cont
   Continuing.
   ^@^@
   ```

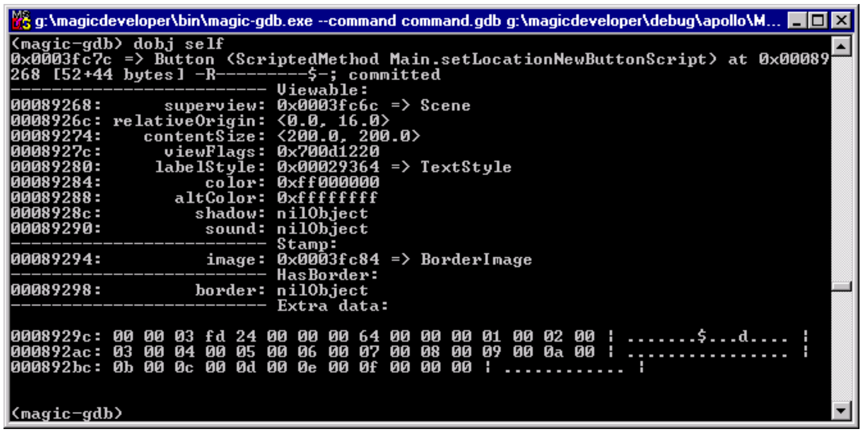
4. In the DataRover 840, move one of the puzzle pieces.

   GDB encounters the breakpoint and breaks into the debugger at the `MakeAMove()` method, and the screen displays the following:

   ```
   Breakpoint 1, MakeAMove
     (
     self=0x3fc7c,
     whichTile=11)
     at Puzzle.cpp:487
   ```

5. At the GDB command prompt, dump the object referenced by the `MakeAMove()` method by entering the command **dobj self**.

   The screen displays the following:



g:\magicdeveloper\bin\magic-gdb.exe --command command.gdb g:\magicdeveloper\debug\apollo\M...

```
(magic-gdb) dobj self
0x0003fc7c => Button (ScriptedMethod Main.setLocationNewButtonScript) at 0x00089
268 [52+44 bytes] -R---------$-; committed
------------------------- Viewable:
00089268:      superview: 0x0003fc6c => Scene
0008926c: relativeOrigin: <0.0, 16.0>
00089274:    contentSize: <200.0, 200.0>
0008927c:      viewFlags: 0x700d1220
00089280:     labelStyle: 0x00029364 => TextStyle
00089284:          color: 0xff000000
00089288:       altColor: 0xffffffff
0008928c:         shadow: nilObject
00089290:          sound: nilObject
------------------------- Stamp:
00089294:          image: 0x0003fc84 => BorderImage
------------------------- HasBorder:
00089298:         border: nilObject
------------------------- Extra data:

0008929c: 00 00 03 fd 24 00 00 00 64 00 00 00 01 00 02 00 ¦ .......$...d.... ¦
000892ac: 03 00 04 00 05 00 06 00 07 00 08 00 09 00 0a 00 ¦ ................ ¦
000892bc: 0b 00 0c 00 0d 00 0e 00 0f 00 00 00 ¦ ............ ¦


(magic-gdb)
```

   Notice that the object information displayed in GDB is the same as the information displayed by the Dump Object window in Microsoft Visual C++.

6. At the GDB command prompt, exit the debugger and disconnect from your DataRover 840 by entering the command **quit**.

GDB does not properly display statically initialized global variables. For example, if you code the following, GDB will not correctly display the value of `string`:

```
char *string = "Hello World";
MyFunction()
{
  dosomethingwith(string);
}
```

The Magic Developer extensions to GDB

The following table summarizes the Magic Developer command extensions that are available in GDB:

Command	Description
`dobj`	**Displays the contents of an object.**
`dloc`	**Displays information about an object locator.**
`pobj`	**Displays the address of an object.**
`cdump`	**Lists all clusters or displays the contents of a cluster.**
`hdump`	**Dumps a heap.**
`set extra-data-max`	**Sets the maximum number of bytes of extra data displayed by** `dobj`**.**
`cnum, getclass`	**Displays the name for a given class number.**
`getindexical`	**Converts between indexical names and numbers.**
`getobject`	**Converts between object names and numbers.**
`dx`	**Disables Debugger() breaks.**
`dx-`	**Re-enables Debugger() breaks.**
`ir`	**Display contents of single register.**
`set fields-as-indexicals`	**Displays the object fields with the indexical name.**
`set magic-step`	**Enables or disables stepping to the target of method dispatches.**

Command	Description
`set magic-xfile`	**Designates X file to use for Magic Cap class information.**
`check-xfile`	**Compares the date of an x-file with the build date.**

In addition, the following commands are available when debugging a remote Magic Cap device:

Command	Description
`autoquiet`	**Enables automatic quieting of sounds when Magic Cap stops at a breakpoint.**
`autoquiet-`	**Disables automatic quieting of sounds when Magic Cap stops at a breakpoint.**
`quiet`	**Manually squelches the sound currently in the Magic Cap sound buffer.**
`unquiet`	**Manually restarts the sound in the Magic Cap sound buffer.**
`set mips-dcache`	**Enables or disables the data cache.**
`mips-dcache-exclude`	**Excludes an address range from the data cache.**
`mips-dcache-include`	**Includes an address range in the data cache.**
`set save-breaks`	**Specifies whether to discard all the breakpoints when the remote device reboots.**
`xfile-path`	**Sets the current package X file search path.**
`set logname`	**Sets a different name for the log file.**

Using the GDB online help

GDB contains an online help system that you can access by entering the command **help** at the (*magic-gdb*) prompt. In addition, you can access help for the Magic Developer extensions by entering the command **help magic** at the command line prompt.

To display detailed help for a specific command, enter **help** followed by the command name at the (*magic-gdb*) prompt. For example, to display help on the *dobj* command, enter the following at the prompt:

```
help dobj
```

6

Object Tools

Magic Cap software is constructed with conventional software development tools and some unique tools developed by Icras, Inc.. These tools implement the Magic Cap object model by preparing source files for the C/C++ compiler and by binding together the compiled package.

From a programmer's point of view, these tools implement most of what is object-oriented about software development for Magic Cap. The planning and design process usually involves building a Magic Cap package from existing classes. The actual C programming you'll do occurs when you have to develop new classes or override the methods of an existing class.

Object model information is specified in two kinds of source files. A **class definition** file contains descriptions of class templates, which can include method definitions that are implemented in separate C files, and is identified by a `.cdef` suffix. An **instance definition** file describes the static objects defined by a package at build-time, and is identified by a `.odef` suffix. See "Class definition syntax" on page 84 and "Instance definition file" on page 92 for descriptions of the files' syntax.

Using or modifying existing classes has several benefits. It saves implementation effort and it reduces package memory resource requirements since the Magic Cap classes are stored in system ROM and special-purpose classes you develop must be stored in RAM.

Terminology overview

In object-oriented programming systems, a **class** is a description of the data and behavior of a set of objects. An **object** is a combination of data structures and procedures that act on that data. A class represents an agreement between a developer and a development system about how those objects will behave when it is called upon to perform some action. Magic Cap classes are defined in a **class definition file**.

Magic Cap further distinguishes between two parts of a class definition: *what* a class can do and *how* it accomplishes this. A class's **interface** represents the first part. It describes the kind of data an instance of a class can hold and the actions it can perform. It is like a group of data structure declarations and function prototypes in a structured programming environment. An **operation** refers to the part of a class's interface that describes the actions it performs. A special case of an operation is an **attribute** which describes how to manipulate the data stored by an object. Attributes usually provide access to fields (see below), and their code is usually generated automatically within constraints specified in the interface.

For a given class interface, a programmer can supply an **implementation** of a class's data storage structures and the code that performs its work or, in some cases, the programmer can indicate that these should be supplied by other classes. This implementation is divided into **fields** that store the data, or instance variables, of a Magic Cap object and **methods** that comprise the code that manipulates the data in a field or performs some action in an object. Fields and methods can take different forms to give a developer flexibility in software design.

Classes can be defined in terms of other classes through a process called **inheritance**. A group of related classes derived from a common source is called a **class hierarchy**.

A class is a static definition and by itself it doesn't do anything. At runtime, Magic Cap can create an instance of a class called an object that performs the actions advertised by the class. In Magic Cap, we often think of objects in two places. The instance definition file contains a textual representation of the **static** objects that the object compiler creates for a package. At runtime a package may create **dynamic** objects as well. For example, whenever the user touches *new* in the Notebook, a new notebook page is created dynamically.

A Magic Cap package is a collection of objects that are organized into an object hierarchy. This hierarchy is different from but related to the class hierarchy used to organize class definitions. See "Object hierarchy" on page 94 for more information.

Object syntax

Class and object definition files are written in a simple language with a small set of syntax rules. Because the tools perform minimal compile-time checking, you should adhere to the naming rules described in the following sections.

A definition file is composed of a series of **statements**. Each statement is terminated by a semicolon.

Two forms of comments are recognized, both borrowed from C and C++.

- Any text between the delimiters `/*` and `*/` is treated as comments and ignored.
- `//` indicates a single line comment. Text to the right of this delimiter is treated as a comment and ignored, up to the end of the line.

There is a simple conditional compilation facility that mimics the C preprocessor. The supported statements are `#define`, `#ifdef`, and `#endif`:

```
#define identifier token-sequence
#ifdef identifier
#endif
```

In subsequent instances, the identifier string will be replaced by the token sequence.

Class definitions may be included in other class definition files, and must be included in object definition files. The **read** statement provides this feature (Although the included file is specified by its `.cdef` source name, it is actually a compiled form with the extension `.cx` that is read). You must include the class definitions for any class that you refer to (as a field type or base class) in defining your own classes.

```
read "fileName";
```

Definition file statements can use a C-like numeric expression syntax. Standard operators like + and - as well as parentheses are supported.

Class definition file

The class definition file contains descriptions of classes that are unique to a given package. The name for this file is usually derived by taking the package's name and appending the `.cdef` suffix to it. Class definition files are compiled during the package build process. Syntax for specifying fields, attributes, operations, and more is described in the following pages.

If a package doesn't define any new classes, it doesn't need a class definition file. Packages can have more than one class definition file; it's a good idea to derive the file name from the class name, for example `ChessPiece.cdef`, `Rook.cdef`, `Bishop.cdef`. The latter two would of course include `ChessPiece.cdef,` since they define classes which inherit from it.

The definition of a new class includes the following:

Item	Description
`class name`	**Each class has a name to associate instances of it with the class definition.**
`inheritance`	**Classes are derived from superclasses and inherit their interfaces and implementation.**
`fields`	**Classes store their data in fields.**
`attributes`	**Attributes are interfaces for special operations that manipulate data in a class.**
`operations`	**Operations describe what you can do to an object.**
`methods`	**Methods are code that implement a class's operations and attributes.**
`overrides`	**Overrides are operations and attributes inherited from some superclass, but re-implemented in the current class.**

The interface part of the class definition specifies the name, arguments, and return type of each operation and attribute of a class. The implementation part defines the fields and methods of a class. The fields define the types for the data holders of a class and the methods define the code or algorithms for the procedures of a class.

To implement the methods of a new class, you must provide source code for the following:

- methods for new operations
- methods for operations you override

The source code for these methods are stored in C files separately from the class definition file. "Method implementation" on page 91 describes how to structure this source code.

Internally, Magic Cap identifies classes and operations by number. If you want, you can manually assign these numbers in the class definition file to prevent Magic Cap from randomly reassigning these numbers between one version of a package and the next.

Class definition syntax

Classes are defined in class definition files like this:

```
define class className;
   // inheritance part
   // interface part
   // implementation part
end class;
```

The first statement in a class's definition specifies the name used to identify it to its subclasses and for objects that are instances of it.

The interior portion of a class definition contains three parts. The inheritance part determines the mechanisms by which one class is derived from other classes. The interface part describes the attributes and operations of the class. The implementation part provides the data handling for the classes fields and the algorithms for its methods.

Inheritance

Magic Cap is like other object-oriented systems in that it has a class hierarchy that organizes classes according to their functionality. The mechanism that enables this system of organization is called **inheritance**. When one class inherits from another class, it inherits its fields, methods, attributes, and operations. It is called a **subclass** of the first class, which is also called a **superclass**.

Classes you create can be derived from other classes through inheritance. Most classes are ultimately derived from the Object class.

There are two ways in which one class can inherit from another. The syntax for enabling this process is the `inherits from` statement. A class definition can contain multiple `inherits from` statements that specify different superclasses.

The various forms of the `inherits from` statement can contain more than a single superclass. You should use a comma to separate superclasses within a single statement.

```
inherits from superclassName;
inherits from superclass1, superclass2;
```

The `inherits from` statement indicates the superclass from which a subclass inherits its attributes, operations, fields, and methods. Operations and attributes in the subclass that need to be changed from what is defined in a superclass can be overridden with the `overrides` statement described below.

Normally, a class will inherit from a superclass that is itself ultimately derived from the Object class. This kind of superclass is sometimes called a **flavor** class.

Mixin classes precisely define their relationship with other classes in the class inheritance scheme. Mixin classes provide the convenience of holding a group of features in a class that can be used in combination with other classes while it is not tied to a specific place in the class hierarchy. This can be more flexible than taking a general purpose class and overriding what you don't need.

Flavor classes must inherit implementation from exactly one flavor and from any number of mixins. They must inherit their interface from at least one flavor and any number of mixins. Mixin classes must "mixin with" exactly one flavor class. They can inherit their interface or implementation from any number of mixin classes. See "Mixin classes" on page 87 for more details.

```
inherits interface from superclassName;
```

This form of inheritance is useful in forcing a class to match the interface of another class and completely provide a new implementation for each attribute and operation of the inherited class. The `inherits interface from` statement has a performance benefit by removing the need for Magic Cap to search for the implementation of a class's methods in the interface superclass. Another benefit of this statement is to provide more type checking during class development.

Extra data

A class that uses the extra data portion of instances to store data whose size varies from one object instance to another declares the fact in the class definition with the `uses extra` keyword.

```
uses extra;
```

In general, when a class specifies `uses extra`, this means that it will be responsible for managing some number of raw bytes whose structure, if any, is private to the class. It's illegal to store object references in such a region, because the Magic Cap runtime (especially the garbage collector) routines have no way to ensure their continued consistency with the rest of the system.

```
uses extra list: type;
uses extra list: type, weak;
```

This class uses the extra data portion of object instances to store a "structured list" whose elements are all of the same type and size. Specifically, if the element type is Object, the array elements are object references and can be maintained by the runtime.

The `weak` keyword indicates that the members of the list are not owned by this object, and so should not, for instance, be automatically deleted when this object is destroyed.

Previous versions of Magic Cap restricted how objects' extra data could be used: only one class in a class hierarchy could use the extra data portion of object instances. The Rosemary release of Magic Cap relieves this restriction with the concept of **shared extra data**.

A class that uses the extra data portion of instances, but also wants to allow subclasses to store their own extra data, declares the fact in the class definition with the `shares extra` keyword. The `ExtraUsed()` method will be called to determine how much of an instance's extra data is maintained by this class.

```
shares extra;
```

A subclass of this class that also wishes to use the extra data portion of instances declares this fact with the `uses shared extra` keyword in its class definition.

```
uses shared extra;
```

If this class wishes to allow subclasses of itself use of extra data as well, it will also need to override `ExtraUsed()`. A class that shares the extra data portion of object instances can call `InheritedExtraUsed()` to find the offset into the extra data where the data specific to that class starts.

The Rosemary version of Magic Cap uses this feature to implement the new viewable format. The list of subviews is maintained in a viewable's extra data portion. However, certain subclasses of class `Viewable` also use the extra data portion of object instances for their own purposes. A prime example of this is class `Card`, which stores form item data in the extra data portion of card instances. Class `Viewable` declares `uses extra list: Object` and `shares extra` and class `Card` declares `uses shared extra` so that both classes have access to the extra data.

If your package implements a subclass that uses extra data, you may need to take into account, in the class definition and possibly some methods, the fact that not all the extra data in an instance belongs to your subclass. In particular, once some superclass uses unstructured extra data with `uses extra`, none of its subclasses may declare structured extra data with `uses extra list`.

Abstract classes

A superclass may be an **abstract** class which is a class that cannot have any instances, though its subclasses can have instances. An abstract class can be derived from another abstract class or from a concrete class.

Abstract classes are used to collect attributes and operations into a convenient class that can then be inherited by other classes. They exist to help organize the class hierarchy and not for the immediate purpose of instantiation. Abstract classes include the keyword `abstract` at the top of their class definition.

Abstract classes can contain fields with the `noGetter`, `noSetter` and `noMethod` flags, as described in “Defining attributes” on page 88 and “Defining operations” on page 89.

Mixin classes

Mixin classes are abstract classes that are independent of `Object` and branch off the class hierarchy. The classes that can inherit from a particular mixin class can be limited to a specific list with the `mixes in with` statement.

```
mixes in with flavorClassName;
```

This statement limits the mixin class to `flavorClassName` and any of its subclasses. To make a mixin class applicable to any class, specify `Object` as the class that it can mix with. Since all flavor classes in the class hierarchy are derived from `Object`, this will allow the mixin class to be applied to any subclass of `Object`.

Mixins also provide a way to distribute functionality to disparate areas of the class hierarchy without having to gather them into a common superclass. By not gathering them, the possible overhead of fields provided by mixins is only carried by the classes that require it and *not* by all the subclasses of the common superclass. This can save memory on an instance by instance basis, especially in a memory limited system like Magic Cap.

Defining fields

Fields provide the implementation for storing data in an object. The `field` statement has the following syntax:

```
field fieldName: fieldType[, optionalFlags];
```

Field names should begin with a lower-case letter. The field type can be any data type or class name. The optional flags modify the properties of a field. The following list describes the different options:

Option	Description
getter	A getter is a method for getting the value in a field. This flag generates a getter method for the field. No interface is created for this automatic getter and a separate attribute statement should also be defined for it.
setter	A setter is a method for setting the value in a field. This flag generates a setter method for the field. No interface is created for this automatic setter and a separate attribute statement should also be defined for it.
weak	The weak flag limits the control an object has over the data in a field. When this flag is set, the object referenced by this field will not be copied with the object that controls it, though the reference remains intact in the copy. In addition, when an object instance is destroyed, the object referenced by this field will not be destroyed with the object that controls it. It can be used to control access to an object during wireline encoding. This flag can only be applied to fields with objects. It is rarely used.

Here is an example of a simple field:

```
field bookColor: Unsigned;
```

This specifies a field named `bookColor` that has the type `Unsigned`.

If the field definition includes a `getter` or `setter` flag, the class compiler generates an automatic method for these fields. The name of the automatic getter is derived from the field's name by capitalizing the first letter in its name. The automatic setter attribute also adds the prefix `Set` to the field's name.

If an attribute is new to a class, then a matching attribute statement is required. If an attribute is inherited, then `getter` or `setter` can be used to override the flags of the inherited attribute. In that case, no additional attribute statement is needed in the subclass. This is described in the next section.

Defining attributes

An attribute describes the interface to methods that access a class's data. There are two forms of attributes. The first form provides a well-defined interface for the automatic methods mentioned in the previous section. Here is an example of field and attribute definitions that a class definition might include:

```
field visible: Boolean, getter, setter;
attribute Visible: Boolean;
```

The first line defines a field with its type and special flags indicating that getter and setter attributes are necessary. The second line defines these attributes. It defines *both* the getter operation `Visible` and the setter operation `SetVisible`. Objects can use these methods to get and set the value of the `visible` field. Note the convention that an attribute is named the same as the underlying field, but the first letter is capitalized.

The second form of an attribute is independent of a specific field in a class. For example, attributes can be created that set and return a calculated value instead. The calculation could be based upon a field value (such as returning a percentage based on a field that maintains a fractional value), the system state (such as the current scene or the modem status), or system constants (such as the image of the logo or a system sound). In this case the package needs to provide the implementation of the getter and setter.

Attributes in class definitions are defined like this:

```
attribute attributeName: attributeType, [flags];
```

The following list describes the different attribute options:

Option	Description
noSetter	**The noSetter flag indicates that the class provides no setter method for this attribute but that subclasses could provide one. This flag can only be used with attributes of abstract classes.**
noGetter	**The noGetter flag indicates that the class provides no getter method for this attribute but that subclasses could provide one. This flag can only be used with attributes of abstract classes.**
readOnly	**The readOnly flag indicates that the attribute can return a value but cannot set one, and so neither this class nor its subclasses should provide a method to do so. Subclasses can redeclare an attribute to add a setter.**

Defining operations

An operation describes the interface to a class's method. It is similar to a procedure declaration with a list of flags that control the operation's behavior. Operations in class definitions are defined like this:

```
operation operationName([paramName: paramType])
     [: returnValueType][, flags];
```

This includes the operation's name, its return value and an optional list of parameters with their types. In addition, an operation definition may include a list of flags that control the behavior of operations.

Ordinary methods in a class have one required parameter, known as `self`. The `self` parameter is a reference to the object whose method is being invoked. Since every method requires this reference as its first parameter, this parameter is omitted in the class definition file. However, you must use it in the C file that contains the source code for the class.

```
class operation operationName([paramName: paramType])
     [: returnValueType][, flags];
```

The following list describes flags for operations:

Item	Description
noMethod	The noMethod flag indicates that the class does not provide a method for this operation. Concrete subclasses of a superclass must provide the method by override.
noFail	The noFail flag indicates that the Magic Cap method dispatcher will do nothing and return 0 (if there is a return value) if the object passed to the operation is a nil object. This may provide convenience during debugging. This flag is only useful during package development on a Magic Cap Simulator since all methods are treated as noFail on a DataRover 840.
intrinsic	The intrinsic flag indicates that this operation is called by a direct jump to the method which is faster than the usual method dispatching mechanism. Intrinsic operations can't be overridden by subclasses. There can be no other flags specified with the intrinsic flag.

See also “Exporting interfaces” on page 91.

Package globals

Previous versions of Magic Cap supported class globals. These are no longer supported. Instead, package globals make porting standard C libraries easier.

Defining simple intrinsics

A class can define operations that don’t require an instance of the class in order to execute. These operations are called simple intrinsics. They have the same properties as intrinsics (see above), but they do not require a reference as their first parameter. Simple intrinsics are faster to invoke than other operations but, like intrinsic operations, they cannot be overridden by subclasses.

Simple intrinsics in class definitions are defined like this:

```
intrinsic intrinsicName([paramName: paramType])[: returnValueType];
```

Overriding methods

Once an operation or attribute has been defined in a superclass, a simple `overrides` statement can be used to provide a new implementation in a subclass. The `overrides` keyword can also define an implementation for a previously unimplemented operation, such as for operations that were defined with the `noMethod` flag in a superclass. A class can also use the `overrides` keyword to override an auto-getter or auto-setter method that was defined for a field of a superclass.

To override a method in a class definition, use the `overrides` statement:

```
overrides operationName;
```

This statement indicates that a given class will supply its own implementation for the operation.

Method implementation

Code from files that have been compiled and linked must be extracted and installed into the correct classes. Magic Cap uses a naming convention to decide which routine's code to attach to an operation. The key part of this convention is that operations in the source file are indicated by the class name and operation name connected with an underscore. For example, a C++ function named `TestClass_DoSomething` would provide the code for the `DoSomething` operation of the `TestClass` class.

The implementation code for a class must be compiled with the preprocessor variable `CURRENTCLASS` set to the Magic Cap class name, so that some Magic Cap features implemented as macros will expand properly.

```
#define CURRENTCLASS TestClass
//  code goes here
#undef CURRENTCLASS
```

When an operation is declared or called in C++ source, the first parameter must be the responder, the object whose operation is being called, normally named `self`. This is analogous to the implicit `this` parameter in C++. When an operation is defined in a class definition file, the responder is implied but is never shown.

Indexicals

Indexicals are used for two purposes: they are the only way to refer to an object from C++ code and they are used in object instance definition files to refer to well-known objects.

Indexicals are declared in class definition files using the following syntax:

```
indexical indexicalName : class;
```

The file `Indexicals.xh` is generated by the Class Compiler to define all system indexicals. Its purpose is to allow C++ code to symbolically refer to indexical objects. It is included automatically when you include `MagicCap.h`.

You can use the class `Object` as a general-purpose place holder for the indexical's class.

Exporting interfaces

You can export your indexicals, classes, and operations for use by other packages by building a separate class definition file for them with their interface definitions. You cannot export intrinsics. This will include all the information needed to use them.

Here is an example,

```
define interface name;
  indexical nameIndexical;
  class nameClass;
  operation nameOperation;
end interface;
```

Example class definition file

Here is an example of a complete `.cdef` file:

```
//
//     ImportSample
//
//     A demo of a package that uses the services of
//     another package (ExportSample)
//
//     General Magic Developer Technical Services
//     Copyright (c) 1992-1996 General Magic, Inc.
//     All rights reserved.
//

read "MagicCap.cdef";

define class Greeter;
       inherits from Viewable;

       overrides Draw;
       overrides AutoMove;
end class;

define class Installer;
       inherits from SimpleActionButton;

       field toBeInstalled: Viewable, setter, getter;
       attribute ToBeInstalled: Viewable;

       operation InstallObjectsIntoStamper();
              // Creates an object from a class
              // defined in another package (ExportSample) and invokes
              // an operation on it.  Interpackage operability!
end class;

indexical iClientScene: Scene;
indexical iCantFindExportSampleInterface: Text;

read "ExportSample.cdef";
import ExportSampleInterface or say iCantFindExportSampleInterface;
```

Instance definition file

Each Magic Cap package must have an **instance definition file** whose name ends with `.odef` that describes the objects used by that package. A package can have more than one instance definition file. As part of the process of building a Magic Cap package, you compile the instance definition file into a package.

The instance definition file contains textual representations of the package's *static* objects. These are the objects the package creates when it is loaded into the Magic Cap environment. The package usually creates other dynamic objects at runtime but they are managed by the Magic Cap environment itself.

Packages which define custom classes may contain static objects of those classes. In that case, the instance definition file may need to include one or more class definition files using the `read` statement.

```
instance class instanceID [objectName][ script ][ flags ];
  fieldName: value;
  .
  .
end instance;
```

The instance definition file simply contains a list of object instances. Each object instance definition includes the name of its class, an instance tag and an optional object name. The name can be helpful in the debugging process though it does incur some system overhead. Objects in an instance definition file do not have to be in any special order.

The instance tag is a bookkeeping device for organizing instance definitions in an instance definition file. The instance tag is a symbolic value, and is retained through procedures like object dumping.

Instance tags are separated into namespaces, so that different sets of objects can use the same tag but still be able to refer to objects in other sets without conflicts. When referencing an instance that was defined with a symbolic tag, the object name is no longer specified:

```
superview: (Scene deskScene);
```

The interior of an instance definition contains a list of field initialization statements with the name of the field followed by its value. These correspond directly to the fields in the object’s class or superclass and must use the same order found in the class definitions. The different values that a field can have are described in the next section.

The instance definition is finished with an `End Instance` statement.

Object dumping

The Magic Cap Simulator can translate objects from the Magic Cap runtime environment to instance definition files. This process of decompiling is called *object dumping*. Dumping is a vital feature for moving between textual and live binary representations of objects. You can also convert objects directly from Magic Cap to text one at a time.

Magic Developer provides a useful way to develop an instance definition file by writing a series of instance definitions. First, build a Magic Cap package with Magic Developer, then modify the objects in construction mode and finally dump them back into Magic Developer. While this is a good technique for developing instance definitions, it does remove any comments you may have stored in your instance file. Place your comments carefully in the instance definition file so that you can easily fold in your modified instance definitions.

Object dumping with the Magic Cap Simulator preserves the instance tags used in instance definition files. When a new object is created at runtime and then dumped with the Magic Cap Simulator, it will be dumped with the tag `unnamed`*`n`*.

See “Dumping single objects” on page 54 and “Opening a copy of the log file” on page 64 for more information about dumping a single object. See “Modifying viewable objects with construction tools” on page 28 for a description of dumping a package.

Object hierarchy

The objects in the instance definition file are connected in a tree-like hierarchy by references to one another. At the top of this hierarchy is the `SoftwarePackageContents` object. Every other object in the instance definition file must be referred to by at least one other object and ultimately referred to by `SoftwarePackageContents` or by an indexical.

Field initialization statements

Each object can contain any number of fields. The instance definition file lists the values that these fields have when the package is loaded.

```
  fieldName: value;
```

Each field initialization statement has a pair of tokens separated by a colon and terminated with a semicolon. The token on the left is the name of the field beginning with a lower-case letter. The token on the right is the value of the field. Each value has a *type* that indicates the kind of data that the field can contain. For example, one field may contain a floating point value and another may contain an indexical.

The following sections describe the different types a field value can have.

Object reference

An object reference takes the following form:

```
(ClassName instanceTag)
```

`ClassName` is the name of the class that the object is instantiated from. *`instanceTag`* should be unique within the namespace.

The object reference information in the field initialization statement should match the information used to define the instance. For example,

```
target: (Greeter MyHelloWorld);
```

Indexicals

Indexicals allow you to refer to standard system objects, important objects in your package or dynamic objects like the current scene. Standard system objects and dynamic objects are listed in `Indexicals.h`. Indexicals can be a reference to an object or a list of objects.

Indexicals are initialized in object definition files with this syntax:

```
indexical indexicalName = (ClassName nameTag);
```

The object list of standard system objects is also called the system root list.

To refer to an indexical in a field of one of your package's objects, you would have a line like the following:

```
labelStyle: iBook12;
```

Operation number

Some Magic Cap classes, like `Control` and `AttributeText`, define fields that require operation numbers as values. System operation numbers are defined for the operations in system classes. The system operation numbers are listed in `OperationNumbers.xh`.

Package operation numbers are assigned to all package-defined operations when a class definition file is compiled. These package operation numbers are stored in the package's `PackageOperationNumbers.xh` file. Magic Developer has a separate instance of this file for each DataRover 840 platform.

For every system or package operation number, a constant of the form `operation_OperationName` is defined. To specify an operation number as the value for a field in instance definition file, use this constant. For example

```
operation: operation_ShowOrHide;
```

Files

A field can extract its value from a file with the `include` keyword. For example, it may be more convenient to store the data for an image in a separate file and refer to it indirectly. The include keyword has three syntax variants.

```
data: include ‘fileName’;
data: include ‘fileName’ start_offset;
data: include ‘fileName’ start_offset:end_offset;
```

The first form includes an entire file as the value for a field. The second form uses the *begin* value as an offset into the file. The third form uses both *begin* and *end* to limit the portion of the file to extract. The end offset is not inclusive. For example,

```
data: include ‘table’ 0:32;
```

includes bytes 0 through 31 of the file `table`.

long

The following lines supply values for fields that are 32 bits in length.

```
myLong: 6;
viewFlags: 0x11005200;
```

The first value is an integer number and the second value is a hex value that is 32 bits in length.

short

The following line supplies a value for a field that is 16 bits in length.

```
myShort: 6.s;
```

byte

The following line supplies a value for a field that is 8 bits in length.

```
myByte: 6.b;
```

Booleans

The following line supplies a value for a field that is 1 bit in length.

```
autoActivate: true; // or false
```

Strings

Strings can include ASCII text, special characters or Unicode constants.

```
name: 'Hello World';
```

Strings can include the following escape characters:

String	Description
\n	**new-line**
\t	**tab**
\’	**single-quote**
\\	**back slash**
\udddd	**Unicode constant**

The best way to include text in a package is with a text object. Fields that store Text objects may be specified in line, however, so the definition above is exactly equivalent to

```
name: (Text HelloWorld);
instance Text HelloWorld;
           text: 'Hello World';
end instance;
```

Fixed

A Fixed value is 32 bits in length with 16 bits of integer and 16 bits of fractional data.

```
width: 1.0; // hex equivalent: 00010000
```

Hexadecimal data

The object compiler accepts both the `$` and `0x` conventions to indicate a string of characters to be interpreted as hexadecimal.

```
object: $00A6E27C;
```

or

```
object: 0x001A5000;
```

Extra data

Many objects store unformatted data as hex strings in their extra data (the variable length portion of an object that can exist in addition to its fields). Objects like images and cards often have extra data.

```
data: $0400 007F FE40 F700 031F FFFF E4F9 000A \
  1701 FD40 017A 0000 01AA A4FD 000E 0FD0;
```

Pixels

Positional data is specified with pixels. A pixel is the basic image unit of a graphics display.

Pixel values are represented by using angle brackets. The value between the <> is a fixed value that includes a decimal point. For example,

```
height <28.0>; // 28 pixels recommended
```

The *Package Development Guide* contains more information on the Magic Cap imaging model.

Dot

An object can store a location on the DataRover 840’s display with a Dot value. The value is represented as horizontal and vertical coordinates in a coordinate system with the origin in the center of the display.

```
location: <20.0,30.0>;// x-y
```

The location of a Dot is specified in pixels.

Box

A Box is a rectangular region of the Magic Cap DataRover 840’s display. The value is represented as a set of four pixels.

```
region: <5.0, 40.0, 80.0, 90.0>; // left, top, right, bottom
```

Magic Script

Scripting for objects has been changed for Magic Cap Rosemary by replacing the old Magic Script with a new language that is compiled at package build-time. This section describes the script language, the Java virtual machine byte codes its script interpreter uses, and the underlying object format its scripts use.

Editing scripts

You can edit scripts in place in instance definition files. They'll be assembled into their old familiar unreadable format as part of the build process. If your script has syntax errors, you'll hear about them when your object definition file is compiled.

You can use the term `script` when you attach scripts to objects. Example:

```
Instance Button makeLikeThis 'Sounds like'
operation script overrides Action: (script scriptTag);
```

What a script looks like

Here's what a script's skeleton looks like:

```
script scriptTag;
        [script prototype is prototype;]

        script statements
end script;
```

If a script omits its prototype statement, it is assumed to have the same prototype as method Action, which takes a Reference and returns void. Scripts must specify their prototypes if the method they're attached to has a different prototype.

If a script needs to return something, it must include a "return integer" or "return object" statement. Otherwise, a return statement is optional. Scripts can have more than one return statement.

Script statements must end with semicolons.

What a prototype looks like

You'll need to provide prototypes for all the operations you call from your script. And if the method you're scripting has a prototype different from Action's prototype, you'll need to prototype your script too. Method and script prototypes look like this:

```
[(argument-type1, argument-type2, ... ) -> return-type]
```

The allowed types are:

- void
- UnsignedByte
- SignedByte
- UnsignedShort

- SignedShort
- Unsigned
- Signed
- Reference

Here are some examples:

```
[(Reference, Reference) -> void] Takes two parameters (probably self plus
                                  another object), and returns nothing.
[() -> void]                      Takes nothing, returns nothing.
[(Reference, Unsigned) ->         Takes an object and an unsigned number
Reference]                        and returns an object.
[(Reference, Reference,           You're calling PerformWithConfirmation
Reference, Unsigned, Reference,   and don't try to deny it.
Reference, UnsignedByte) -> void]
```

Stack operations

Most script operations expect to find parameters stored on a push-down stack in memory, and many return a result value by leaving it on the stack. So the most primitive operations simply move values onto and off of the stack.

```
push constant
push (ClassName objectInstanceTag)
push iIndexicalName
push nilObject
push self
push integer
push 0xlongHexValue
```

Use push to push objects and numbers onto the stack.

Refer to objects using the format your object definition file uses. Push a named object onto the stack like this: `push (EditWindow chooseANameWindow)`. You can also use indexical names: `push iCurrentUser`.

You can also push numbers onto the stack. Use the form `push 27` to push a byte or a short onto the stack. Prepend `0x` before your number in hex to push a long onto the stack.

```
pop
```

Pop with no arguments discards the top item on the stack.

```
pop into variable index
```

Pop the top of the stack into the variable numbered *`index`*.

```
push variable index
```

Push the variable numbered *`index`* onto the top of the stack.

```
push argument index
```

Push the given argument to the script onto the stack.

If a script takes more than just self as an argument, the additional arguments are stored in script variables. For example, if you've scripted `Touch` for some object, you'll get the `touchInput` parameter by calling `push argument 1`.

`push argument 0` is equivalent to `push self`.

```
dup
```

Duplicate the top item on the stack.

```
swap
```

Swap the top two items on the stack.

```
copy into variable index
```

Copy the top item on the stack into the numbered variable.

```
goto label
```

Jump to the named label. Labels can't include white space.

```
call MethodName prototype
call intrinsic_IntrinsicName prototype
```

Call a Magic Cap method. The method has the given prototype. Before calling the method, make sure you've pushed the required arguments onto the stack in the correct order. Push in order from left to right.

After the call completes, the result will be on the top of the stack. The arguments are removed from the stack.

Suppose you need to write a script that does the equivalent of this rather common Magic Cap statement:

```
personWhoWontBeMissed = At(gotThemOnMyList, 3)
```

You'd first push the arguments to At onto the stack, then call At. Your script would look like this:

```
push (ObjectList gotThemOnMyList);
push 3;
call At [(Reference, Unsigned) -> Reference];
```

If you're calling an intrinsic, you must refer to it as `intrinsic_IntrinsicName`. To make your script honk, you'd write this line:

```
call intrinsic_Honk [() -> void];
```

Comparisons

```
if equal, goto label
```

If the top two items on the stack are equal, branch to the named label. Otherwise, continue with the next instruction. This statement removes the top two items from the stack.

```
if not equal, goto label
```

If the top two items on the stack aren't equal, branch to the named label. This statement removes the top two items from the stack.

```
if 0, goto label
```

If the top item on the stack is equal to zero, branch to the named label. The top item is removed from the stack.

```
if not 0, goto label
```

If the top item on the stack is not zero, branch to the given label. The top item is removed from the stack.

```
if < 0, goto label
if <= 0, goto label
if >= 0, goto label
if < 0, goto label
```

Compare the top item on the stack to zero. If the comparison is true, branch to the given label. The top item is removed from the stack.

```
if nilObject, goto label
```

If the top item on the stack is `nilObject`, branch to the given label. The top item is removed from the stack.

```
if not nilObject, goto label
```

If the top item on the stack is not `nilObject`, branch to the given label. The top item is removed from the stack.

Arithmetic and logical operations

```
add
```

Add the top two items on the stack. The two items are replaced on the stack by the result.

```
subtract
```

Subtract the top item from the second item on the stack. The two items are replaced on the stack by the result.

To compute a - b, you'd write these script lines:

```
push a;
push b;
subtract;
```

```
multiply
```

Multiply the top two items on the stack. The two items are replaced on the stack by the result.

```
divide
```

Divide the second item on the stack by the first item. The two items are replaced on the stack by the result.

To compute a / b, you'd write these script lines:

```
push a;
push b;
divide;
```

```
remainder
```

Divide the second item on the stack by the first item and return the remainder. The two items are replaced on the stack by the result.

To compute a mod b, you'd write these script lines:

```
push a;
push b;
remainder;
```

```
negate
```

Change the sign of the number on the top of the stack. The number is replaced by its opposite on the stack.

```
bitwise and
```

Compute a bitwise and of the top two items on the stack. The top two items on the stack are replaced by the result.

```
bitwise or
```

Compute a bitwise or of the top two items on the stack. The top two items on the stack are replaced by the result.

```
bitwise xor
```

Compute a bitwise exclusive or of the top two items on the stack. The top two items on the stack are replaced by the result.

```
return
return object
return integer
```

Clear the stack and return. *`return object`* and *`return integer`* return the item on the top of the stack and claim that the top is of the given type.

If your script doesn't need to return anything, you don't need to include a return statement. If your script's prototype says it returns something, you must end your script with return integer or return object.

7

Package Localization

Magic Cap provides support for internationalization based on Unicode. Icras, Inc. can create localized versions of Magic Cap for different international markets while Magic Cap package developers can develop localized versions of their packages for these different markets.

The key to developing a localizable Magic Cap package is to isolate localizable elements so that the package can be localized for different languages without the need for source code modification. Magic Developer includes localization tools that help package developers separate the tasks of feature development and localization.

Preparing packages for localization

Most of the work done to support localized Magic Cap packages is done by the system software and Magic Developer's localization tools. Here are a few guidelines to developing localizable Magic Cap packages:

- Don't use C or Pascal strings in your package's source code for user-visible text. Instead, use text objects because they can be easily identified and localized with Magic Developer's conversion scripts.
- Don't make assumptions about the specific location of your package's viewable objects. You may need to move and resize viewable objects for convenient reading.
- Don't make assumptions about the specific visual appearance of your package's viewable objects. You may find it necessary to change the images for viewable objects.

Localization files

The process of localizing a Magic Cap package involves several different files:

- An *object instance definition file* describes the static objects used by a package. This includes all the text objects and the size and placement of viewable objects.
- A *phrase file* describes modifications made to an object instance definition file. This file is combined with other Magic Cap objects in the package build process to create a localized version of a package.

Using phrase files

Phrase files let you modify strings and other objects in your object definition files to accommodate a different locale. For example, if one image is appropriate for the Japanese locale and another image is appropriate for the US locale, you may place each image in a separate phrase file and it will be built into the package for that locale.

Phrase files are named `*Locale*.Package.Phrases`, where *`Locale`* specifies the name of the country for which the package is targeted. For example, `Japan.Package.Phrases` and `USA.Package.Phrases` are typical phrase files.

Compiling with phrase files

If you are creating a package for use only in the United States, you do not need to compile with a phrase file, although you can optionally specify one. If you are using the package in any other country, you must compile with a phrase file.

Magic Developer provides two ways to prevent a package from compiling with a phrase file:

- Specify the following switch in the makefile:

```
-no-require-phrases
```

- Specify the following statement in the *`Locale`*`.Package.Phrases` file:

```
dont require phrases for textual fields
```

To allow a phrase file when you compile, remove or comment both of these statements. Magic Developer then uses the phrase file associated with the locale you specify when you build the package. See “How to build a package” on page 17 for information about choosing a locale when you compile.

Phrase file syntax

A phrase file describes the modifications that you want to make to an instance definition file—an `.odef` file. The phrase file contains three-line records separated by blank lines. Following is the syntax for the phrase file entries:

```
phrase for ObjectName field FieldName
replace OriginalValue
with NewValue;
```

The first line specifies the name of the object and field of the instance definition to which the record applies. The second line specifies the original value of the object you want to modify. The third line, and any other lines, if necessary, specify the new value for the modified object.

For example, the following phrase replaces the value of the field `text` of the object `packageSceneInformation` with the value of the `with` string:

```
phrase for packageSceneInformation field text
 replace 'Yer guide to AhoyWorld\nThem seamen shout \u2018ahoy\u2018.'
 with 'About Hello World\n\u2018Hello World\u2019 is landlubberese
      for \u2018 Ahoy.\u2019';
```

Including another phrase file

Use typical C++ syntax to include an additional phrase file. For example, to include the file `Resizing.phrases`, enter the following line in your main phrase file:

```
#include "Resizing.phrases"
```

This line has the same effect as placing the actual content of `Resizing.phrases` where this `#include` line appears. The path of the phrase file is relative to the including file's path. Files can be nested, so an included file can also contain an `#include` line.

Localizing text

Much of the effort in localizing a Magic Cap package is in translating text strings from the source language to the target language. Localizing the text in a Magic Cap package is a two-step process:

- Create a phrase file for the target locale.
- Translate the strings in the phrase file.

The rest of this section discusses these procedures.

Creating a phrase file

When you compile a localized package with an empty phrase file, Magic Developer displays an error for each text string that you need to replace. The error message looks similar to the following:

```
D:\MagicDeveloper\samples\HelloWorld\Objects.odef (22) :
Error: Missing phrase file entry for name of object 'Main.contents'.

phrase for Main.contents field name
replace 'HelloWorld'
with '<L>HelloWorld';
==== If the entry should not be translated, add the following lines.
phrase for Main.contents field name
replace 'HelloWorld'
with 'HelloWorld';
```

The error message shows two possible replacements for each text string:

- A new string that begins with an <L> to indicate that it must be translated.
- An identical string that you use if the original text string does not need translation.

You can copy and paste the appropriate replacement for each string into your empty phrase file. Each entry in the phrase file must have three lines: the `phrase` statement, the `replace` statement, and the `with` statement.

For example, suppose you want to localize the HelloWorld sample package for a release in Japan:

1. Launch Microsoft Visual C++ and open the workspace file for the HelloWorld package.

2. Choose Build▶Set Active Configuration, then specify the configuration.

For example, choose Apollo Japan Release.

3. Choose File▶Open, then open the phrase file for Japan, `Japan.Package.Phrases`.

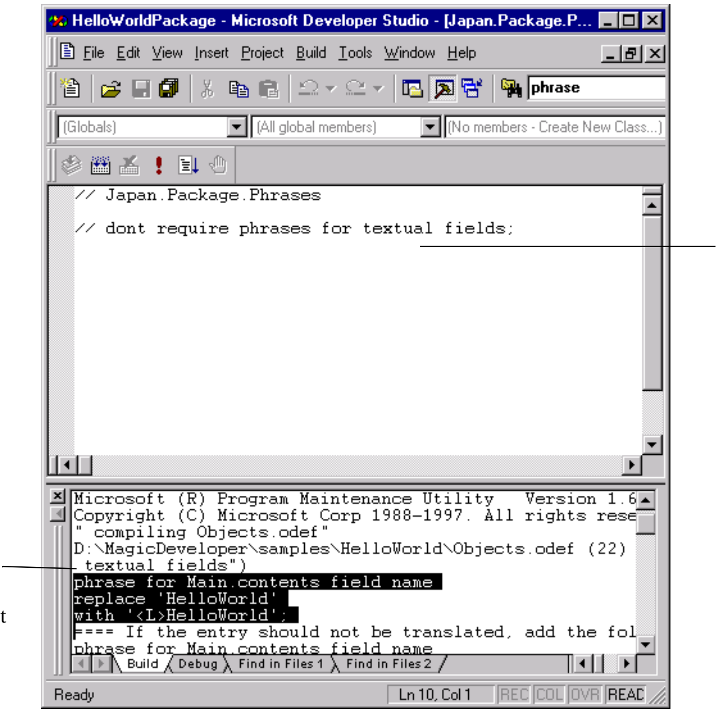
4. Place a `//` comment before the following line:

```
dont require phrases for textual fields
```

5. Build the package by choosing Build▶Build, or by pressing *F7*.

Microsoft Visual C++ displays an error for each missing phrase in the Build window.

6. Create a new phrase file by copying the replacement strings from the Build window and pasting them into the `Japan.Package.Phrases` file.


HelloWorldPackage - Microsoft Developer Studio - [Japan.Package.P...
File Edit View Insert Project Build Tools Window Help
phrase
(Globals)
(All global members)
(No members - Create New Class...)
Build
Debug
Find in Files 1
Find in Files 2
Ready
Ln 10, Col 1
REC
COL
OVR
READ

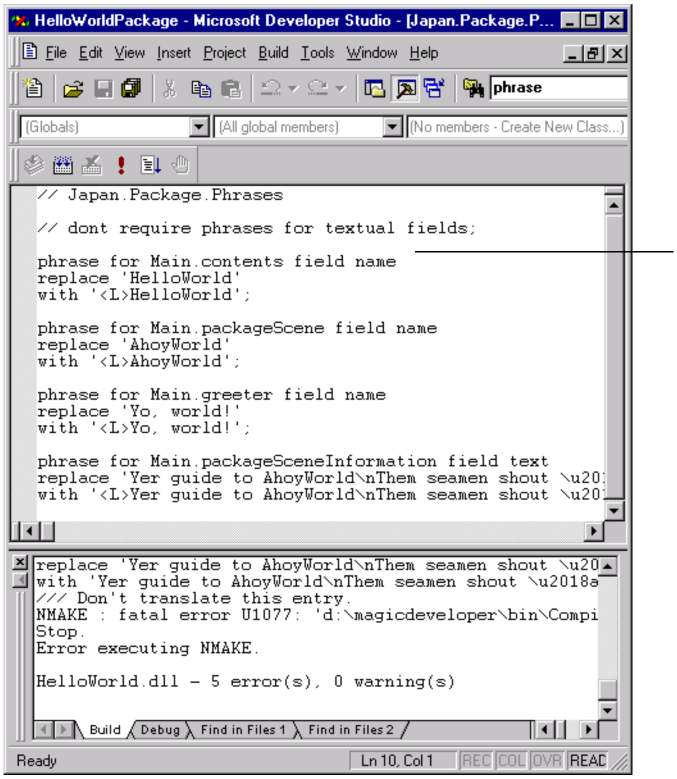
```
// Japan.Package.Phrases

// dont require phrases for textual fields;
```

```
Microsoft (R) Program Maintenance Utility   Version 1.6
Copyright (C) Microsoft Corp 1988-1997. All rights rese
" compiling Objects.odef"
D:\MagicDeveloper\samples\HelloWorld\Objects.odef (22)
 textual fields")
phrase for Main.contents field name
replace 'HelloWorld'
with '<L>HelloWorld';
==== If the entry should not be translated, add the fol
phrase for Main.contents field name
```

...then paste the replacement phrase into the new phrase file

Select a replacement phrase in the Build window and copy it to the clipboard...

When you are finished, the new phrase file should look similar to the following:


HelloWorldPackage - Microsoft Developer Studio - [Japan.Package.P...
File Edit View Insert Project Build Tools Window Help
phrase
(Globals)
(All global members)
(No members - Create New Class...)
Build
Debug
Find in Files 1
Find in Files 2
Ready
Ln 10, Col 1
REC
COL
OVR
READ

```
// Japan.Package.Phrases

// dont require phrases for textual fields;

phrase for Main.contents field name
replace 'HelloWorld'
with '<L>HelloWorld';

phrase for Main.packageScene field name
replace 'AhoyWorld'
with '<L>AhoyWorld';

phrase for Main.greeter field name
replace 'Yo, world!'
with '<L>Yo, world!';

phrase for Main.packageSceneInformation field text
replace 'Yer guide to AhoyWorld\nThem seamen shout \u20
with '<L>Yer guide to AhoyWorld\nThem seamen shout \u20
```

```
replace 'Yer guide to AhoyWorld\nThem seamen shout \u20
with 'Yer guide to AhoyWorld\nThem seamen shout \u2018a
/// Don't translate this entry.
NMAKE : fatal error U1077: 'd:\magicdeveloper\bin\Compi
Stop.
Error executing NMAKE.

HelloWorld.dll - 5 error(s), 0 warning(s)
```

The finished phrase file contains all the replacement phrases, indicated with <L> markers

7. Build the package again.

   If you have correctly replaced every phrase, Microsoft Visual C++ displays no more errors.

If you run this package, you'll see the `<L>` strings displayed in the application. These `<L>` strings indicate where the translated text will appear.

Translating the strings

After you create the replacement phrase file, the strings are ready to be translated. When you deliver the file for translation, specify that any string which begins with an `<L>` marker should be localized.

After the strings are translated, convert them into Unicode and then build the package to confirm that you do not receive any errors.

For example, suppose you want to finish localizing the strings in the `Japan.Package.Phrases` file you created in "Creating a phrase file" on page 106:

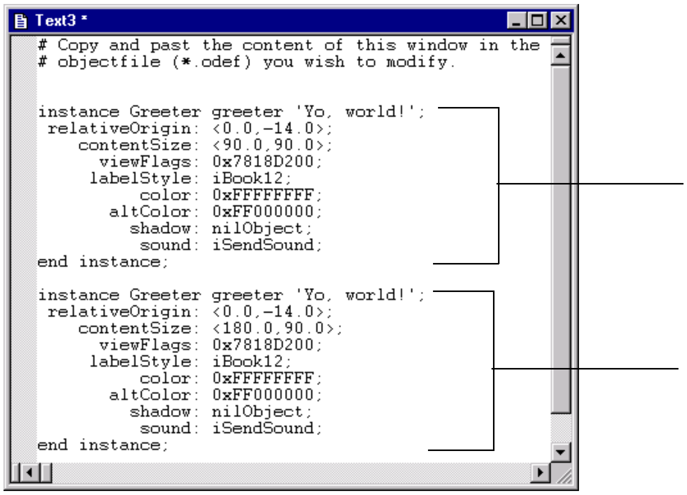
1. Translate the strings marked with an `<L>` in `Japan.Package.Phrases`.
2. Convert the translated strings into Unicode.
3. Build the localized package with the new phrase file.
4. Test the localized package. Make sure you use the correct localized version of Magic Cap Simulator.

Use the `<L>` characters to help you debug your localized package. If a string in your package still contains the `<L>` characters, you know that the string was never translated.

Modifying viewable objects

When you localize a package, you may need to change the size and location of viewable objects, and also the image they display. You can make these changes in Construction mode in the simulator. First dump the original object to the log file so you have a record of it, make any necessary modifications in the simulator, then dump the finished object.

You can use the code you dumped to the log file as the basis for the entries you need to make in the phrase file. For example, suppose you want to resize the `greeter` object in the HelloWorld sample file. When you open the log file, the code for this modified object looks similar to the following:



```
# Copy and past the content of this window in the
# objectfile (*.odef) you wish to modify.


instance Greeter greeter 'Yo, world!';
 relativeOrigin: <0.0,-14.0>;
    contentSize: <90.0,90.0>;
      viewFlags: 0x7818D200;
     labelStyle: iBook12;
          color: 0xFFFFFFFF;
       altColor: 0xFF000000;
         shadow: nilObject;
          sound: iSendSound;
end instance;

instance Greeter greeter 'Yo, world!';
 relativeOrigin: <0.0,-14.0>;
    contentSize: <180.0,90.0>;
      viewFlags: 0x7818D200;
     labelStyle: iBook12;
          color: 0xFFFFFFFF;
       altColor: 0xFF000000;
         shadow: nilObject;
          sound: iSendSound;
end instance;
```

The log file contains code for the original, unmodified object...

...and code for the modified object

You need to change this code so it uses the proper syntax for the phrase file. Each entry for an object modification in the phrase file must have three lines. For example, code for the resized `greeter` object in the HelloWorld sample would look like the following in the phrase file:

```
phrase for Main.greeter field contentSize
replace <90.0,90.0>
with <180.0,90.0>;
```

In this example, the first line specifies the name of the object, the second line specifies the original size of the object, and the third line specifies the new value for the size.

The following procedure shows you how to modify a viewable object for localization:

1. Build a localizable Magic Cap package.

   See “How to build a package” on page 17.

2. Launch the package in Magic Cap Simulator.

   See “How to test a package” on page 18.

3. Display the scene with the viewable object you want to modify.
4. Start construction mode.

   See “Turning on construction mode” on page 38.

5. Dump the unmodified object into the log file.

   See “Dumping single objects” on page 54.

6. Use the move and stretch tools to modify the viewable objects to match the placement and size needed for the new locale.

   See “Move tool” on page 50 and “Stretch tool” on page 50.

7. Dump the modified object into the log file.
8. In Microsoft Visual C++, use the `FixUpLogFile` macro to open a copy of the log file.

   See “Opening a copy of the log file” on page 64.

9. Use the information in the log file to modify the phrase file.
10. Rebuild the localized package.
11. Test the localized package.

Index