Rebol

A programmer's guide

First edition – December 2008

By visiting the site at www.auverlot.fr you can

- Talk with the authors
- Download the example code
- Look for updates and additions

Preface

For many years the answer to the question “Which is the best book to learn Rebol?” has been Olivier Auverlot’s “Programmation Rebol”. As you probably guessed from the title, it’s written in French. Soon after it was published late in 2001, work started on a translation. It was even listed with a publication date on Amazon.com. Sadly, it has yet to see the light of day.

At the beginning of 2007, Olivier published “Rebol - Guide du programmeur”. I hoped for news of a translation; but none came. It seemed that the only option was going to be to read the book in French. So why not translate the book myself?

Translating “Rebol - Guide du programmeur” has been most enjoyable. Olivier’s writing is eloquent and informative. His command of Rebol shines through. I learned a lot whilst translating the book, some Rebol and some French. I hope you will too, though perhaps not so much French as me.

Acknowledgements

I would like to thank Olivier not only for writing such a good book but equally for all the help and support he's provided as I've been working on the translation.

There are many helpful folks in the Rebol community. In particular, Gregg Irwin and Sunanda have been a constant source of friendship and encouragement. Thank you.

Last but certainly not least, I would like to thank Noriati, my wife, and Hannah Sarah, our daughter, for their love, their unquestioning support and their forbearance during the many hours I've sat staring at my computer.

Peter W A Wood

Foreword

What is the purpose of this book?

For five years, Olivier had the pleasure of working for the French computer magazine Login and wrote many articles about REBOL. Sadly Login is no longer published and Olivier felt it both necessary and appropriate to consolidate his work so that the accumulated knowledge would still be available to the Rebol community. Understandably, he initially published his work as “Rebol – Guide de Programmeur”. This book is a translation of the original into English.

The basic idea was simply to regroup articles according to specific subject, but the project turned out to be much larger than that. Many articles required a partial rewrite, others needed to be supplemented and finally, some previously unpublished elements were added. The "remix" has been much greater than anticipated, but it was necessary to ensure that the book has a coherent structure.

In the end, the book turned out to be a highly practical guide to Rebol for programmers.

The diversity of topics makes the book more of a reference manual than a complete introduction. It elaborates on a number of points from Olivier's previous book "Rebol Programmation" published by Eyrolles (ISBN: 2-212-11017-0).

Who is this book for?

This book is primarily aimed at Rebol developers. It was conceived and designed with them in mind, giving the maximum of knowledge across a wide range of topics in a concise form. Most importantly, it can save them a lot of time through the numerous examples provided.

It also meets the needs of developers and students wishing to learn Rebol. Each chapter explains the strong points of the language and then applies that knowledge.

Additionally, it meets the needs of policy makers by outlining the capability of Rebol technology. Various case studies and product presentations are used to study the implementation of the language with the help of examples.

Finally, it is intended for system administrators interested in using Rebol to automate certain tasks (Unix server administration, managing grid computing, etc.). Many parts of this book are devoted to such topics.

How is it structured?

The book consists of seven chapters each built around a theme:

- Chapter 1 is an overview of Rebol by example,
- Chapter 2 deals with the basics of the language,
- Chapter 3 covers graphical interfaces, graphics and sound,
- Chapter 4 describes Rebol's advance network programming features,

- Chapter 5 is aimed at professional users of Rebol. It presents the use of Rebol in the context of e-business,
- Chapter 6 provides advanced information for Rebol developers,
- Chapter 7 is a series of workshops designed to facilitate learning the language through practice.

Contents

Introduction

Chapter 1 – Discover Rebol in an hour

Chapter 2 – The Rebol langauge

Chapter 3 – GUI, graphics and sound

Chapter 4 – Networking and the Internet

Chapter 5 – Rebol for pros

Chapter 6 – Rebol for geeks

Chapter 7 – Practical applications

Table of Contents

Introduction 1

Join the REBOLution ! 1

Programmer, word builder 2

A virtual machine 3

A family of products 3

Rebol/Core 4

Rebol/View 4

Rebol/Command 5

Rebol/SDK 6

Rebol/IOS 6

A network programming language 6

Manipulating information 7

Object programming 7

The main applications written in Rebol 8

Protocols and dialects 8

Rebol and the web 8

Utilities as if it raining! 9

Multimedia and games 10

Downloading and installing 11

Using the console 14

Establishing a working environment 15

Chapter 1 - Discover Rebol in an hour 17

The robotFTP project 17

Project introduction 18

Technical considerations 18

The execution environment 18

The security manager 19

Writing the header 20

The principle datatypes 21

And now, the code! 22

Declaring a function 23

Manipulating URLs 24

Executing FTP commands 25

Defining and using dialects 25

The robotFTP dialect 26

Managing errors 27

Conditional expressions 28

Managing files 28

Completing and testing robotFTP 29

Adding a graphic layer 30

Managing a progress bar 30

Opening a window 31

Integration with the RobotFTP script 31

Summary 32

Chapter 2 – The Rebol language 33

Survival guide 33

Let's write a program 34

Running your script 35

Variables and datatypes 36

Declaring a variable 36

Finding the type of a variable 37

Using Constructors 37

Simple datatypes 38

Complex datatypes 38

Blocks 39

Handling lists 40
Arrays 40
Navigating within a series 41
Accessing an element 42
Adding and removing elements 42
Modifying a series 43
Searching and sorting series 43
Copying and clearing series 43
Control structures and loops 44
Tests in Rebol 44
And else? 45
Multiple choices 46
Loop 46
The for loop family 47
Repeat, until and while 48
A simple game in Rebol 49
Functions and objects 50
Using functions and objects 50
Defining functions 51
Creating objects 53
Parsing and dialects 55
The art of handling character strings 55
Rebol parsing 56
Parsing using a dialect 56
A little parsing 57
Defining a dialect 58
Summary 60

Chapter 3 – GUI, graphics and sound 61
GCS and VID 62
Basic concepts 62
Styles 64
Attributes 64
Style layout 65
A dollar-euro converter 65
Image processing with VID 66
Using images 67

Displaying an image 67
Modifying images 69
Applying special effects 70
The DRAW dialect 72
Let's draw a line 72
Other draw functions 74
Adding text 75
Manipulating images 76
Generating image files 77
Using DRAW dynamically 77
Generating DRAW instructions 77
A little animation 79
Handling events with VID 80
Event-driven programming 80
Default behaviours 80
Tracking events 81
The event object 83
Controlling Windows 84
Managing styles 85
Defining a style 85
Applying a style sheet 86
Modifying style aspects 87
Defining a style's behaviour 89
Rebol and sound 90
Opening and closing a sound port 90
Loading and handling sound samples 91
Playing Samples 93
Graphic display of a sound sample 94
Summary 96

Chapter 4 – Networking and the Internet 97
Using TCP/IP protocols 97
Protocols in Rebol 98
Network configuration 98
Sending and receiving email 99
Accessing web resources 101
And the other protocols? 101

Clients and servers 102

Creating network protocols in Rebol 103

The standard protocols 104

The root protocol 105

Properties of the root-protocol object 106

Methods of the root-protocol object 107

Implementing Echo 108

Developing a gopher protocol 110

Rebol and CGI scripts 113

The overall picture 114

CGI overview 114

How to write CGI scripts in Rebol? 115

Reading parameters 116

Producing dynamics web documents 119

How Magic! Works 120

Installation 120

When the static becomes dynamic 121

Magic! Functionality 123

Library functions 123

Controlling embedded Rebol code 124

Handling MIME types 124

Managing cookies 126

Managing sessions 126

Handling XML documents 128

The problem of interoperability 129

Rebol's integrated parser 129

How to use the data? 130

The guts of the parser 132

A better XML parser 133

Generating XML 135

Using Web Services 136

Introducing XML-RPC 136

Using XML-RPC 136

The Rebol/View plugin 137

Small yet powerful 137

Inserting the plugin in an HTML page 139

Configuration Parameters 140

Cache, proxy and compression 141
Interacting with the Browser 144
Summary 147

Chapter 5 – Rebol for pros 149
Rebol/Command 149
Concentrated power 150
Database access 150
Shell access 151
Using dynamic libraries 152
Data encryption 153
Rebol, CGI scripts and MySQL 153
Introducing the MySQL protocol 154
Downloading and installing the protocol 154
Using the protocol 155
Integration in a CGI script 156
Data encryption 158
Symmetric key encryption 159
RSA encryption 160
Diffie Hellman 162
Signing your document with DSA 163
Rebol/IOS 164
IOS: a concept lacking exposure 164
An ultra-secure system 165
The Rebol/Serve server 166
The Rebol/Link client 168
Reblets 169
Administration tools 172
Managing Rebol projects 174
The Prebol preprocessor 174
Installation and use 175
Including files 176
Evaluation and conditions 178
Summary 179

Chapter 6 - Rebol for geeks 181
Rebol and virtual desktops 181

A work environment 181
The Rebol/View desktop organisation 182
The file index 183
Unix programming with Rebol 183
Displaying data in a console 184
Managing the keyboard 188
Integration with the Unix environment 190
Files and access rights 192
Shell access 194
Inter-process communication 195
Managing Unix signals 199
Interfacing with dynamic libraries 202
Databases with RebDB 205
Getting started with RebDB 205
Tables and fields 206
Manipulating data 208
Putting it to work 209
Editing and saving 211
Publishing your database on the web 213
Summary 215

Chapter 7 – Practical applications 217
Writing a raycasting engine with View 217
What is raycasting? 218
It looks real to me! 218
Starting with the foundations 219
Shaping the walls 220
Finishing with interaction 222
Program your "chat" in Rebol 223
Basic principles 223
Writing the client 224
Setting up the server 226
A MySQL administrative console 228
The specification 229
User identification 229
Creating the main screen 230
Some utility functions 231

The processing functions 232
Writing a reblet for IOS 234
The specification 235
The development cycle 235
Defining a fileset 236
The POST methos 237
The client code 238
Summary 240

Introduction

Rebol (*Relative Expression-Based Object Language*) is the work of Carl Sassenrath who was one of the main creators of the esteemed AmigaOS operating system. In 1995, he decided to work on a new programming language and his efforts came to fruition in 1997 with version 1.0 of Rebol.

Carl's objective was to design a language that is simple to learn, multi-platform and fully adapted to the exchange of information between heterogeneous systems. As such the language is highly capable in the field of network programming. It supports the major network protocols and allows easy handling of sockets. Rebol is an ambitious concept realised in a range of products.

Join the REBOLution !

The intelligence underlying the Rebol concept, based on the simple but powerful observation that with the rise of importance of computers communicating with each other, is that programmers need a new generation of more expressive languages with syntax closer to human language and the ability to use standard protocols to exchange information via TCP/IP.

Modern software must be designed not with the picture of just a single computer in mind but a network of interconnected systems to share data and programs. Like Python, Ruby and some aspects of Java and .NET solutions, Rebol is an ambitious, innovative scripting language. It is much more than a simple programming language, it can be called a meta-language, the heart of a device dedicated to the exchange of information over networks.

Programmer, word builder

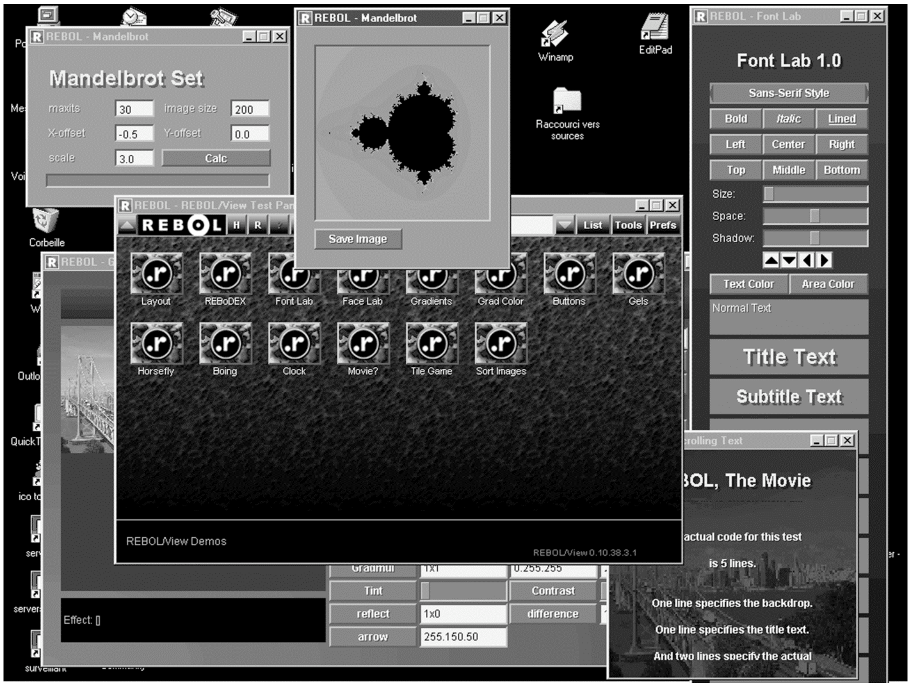
In the conventional sense, Rebol is an interpreted rather than compiled language. The code is not converted into an executable binary or even *bytecode* like Java. In fact, Rebol is an evaluator like Lisp and Scheme. They build a vocabulary from a script and interpret the words you've set as well as those belonging to the standard dictionary of the language. This mode of operation makes Rebol a meta-language: that is a language with the ability to describe, redefine and enrich its own internal mechanisms.

In Rebol, everything boils down to words. Instructions, variables, functions and objects are all words. Some simply provide particular functionality. All these words belong to either a general context (the global context) or a specific context. Just as in a spoken language, a word may have a special meaning depending on the context in which it is evaluated.

Words can also be defined in a dialect which is a kind of mini-language often referred to as a domain specific language or DSL these days. A dialect is a language within a language, to perform a very specialised task. To better understand dialects, we can take engineering as an example. When engineers talk about their profession, a person from outside their discipline often finds it very difficult to grasp the meaning of the conversation. Admittedly, these engineers speak English but enrich it with technical terms that form a dialect known only between themselves! In a Rebol script, dialects allow the description of certain portions of an application by using a specialised vocabulary which can not only be manipulated by a programmer but also an outsider who is not necessarily a developer. With dialects, we can define business rules, graphic interfaces or a sequence of screens during the installation of a program.

A virtual machine

The evaluator is actually a virtual machine which simulates an identical environment regardless of the underlying operating system and hardware architecture. Unlike Java, it is very light weighing in at between 250 and 400 KB depending on the version. It is also very easy to install because it consists of a single executable. No! you are not dreaming, the Rebol virtual machine is entirely contained in a single file. There are no DLLs or libraries to install whatever the operating system. In a few seconds, it is possible to transform any computer into a machine capable of using your applications written in Rebol.



actual code for this test is 5 lines.

One line specifies the backdrop.

One line specifies the title text.

And two lines specify the actual

Figure 0-1. *Running a Rebol application.*

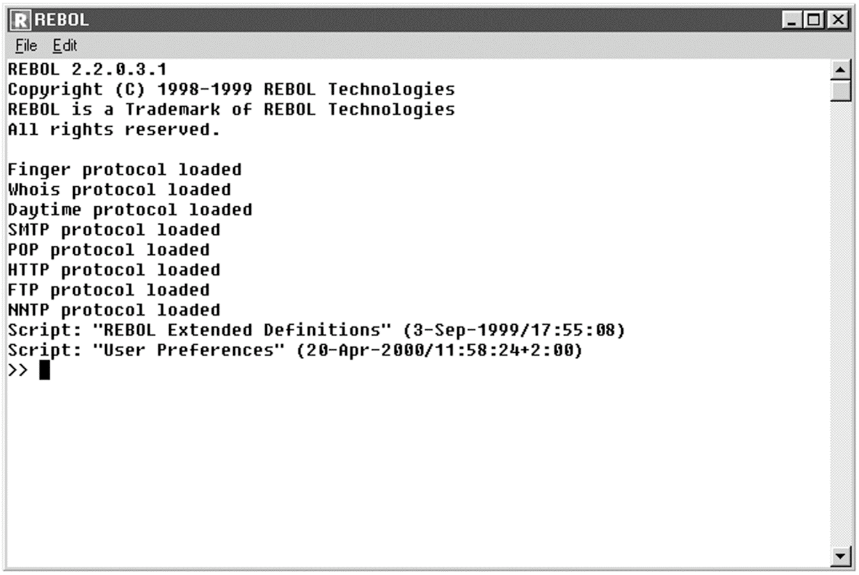
This virtual machine contains the Rebol evaluator, the standard dictionary, TCP/IP support, a garbage collector and security manager.

A family of products

Rebol comes in six versions aimed at different audiences and also targeting different application areas.

Rebol/Core

Rebol/Core is the foundation. This compact evaluator, of around 350 KB, contains the heart of the language. It includes the evaluator, the garbage collector, the security manager, network protocols and the words in the Rebol dictionary. Distributed under a "freeware" licence, this product can be freely downloaded from rebol.com for many operating systems and can be freely used in commercial products.



```
REBOL 2.2.0.3.1
Copyright (C) 1998-1999 REBOL Technologies
REBOL is a Trademark of REBOL Technologies
All rights reserved.

Finger protocol loaded
Whois protocol loaded
Daytime protocol loaded
SMTP protocol loaded
POP protocol loaded
HTTP protocol loaded
FTP protocol loaded
NNTP protocol loaded
Script: "REBOL Extended Definitions" (3-Sep-1999/17:55:08)
Script: "User Preferences" (20-Apr-2000/11:58:24+2:00)
>>
```

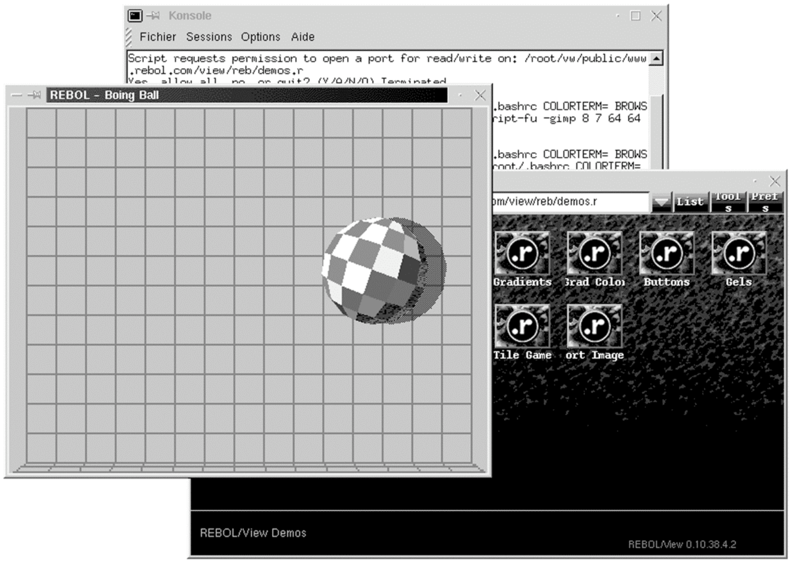
Figure 0-2. *The Rebol/Core console.*

Rebol/Core is not a demonstration version or limited product. It is the perfect tool for developing applications to exchange information, write CGI or system administration scripts.

Rebol/View

Rebol/View is an overlay on top of Rebol/Core which adds the ability to develop graphic applications. Since version 1.3, Rebol/View uses the very powerful AGG graphics library as its display engine. It's possibilities are impressive since AGG provides advanced graphics primitives, excellent display speed, anti-aliasing management, vector fonts and incredible effects which can be applied in real-time (rotation, perspective, size, transparency, colour processing, etc.).

For building user interfaces, Rebol/View incorporates VID (*Visual Interface Dialect*) allowing the definition of screens with a minimum of instructions. The main user interface components are included as standard (button, input field, slider, image, progress bar, etc.). You can also change their appearance by using a style sheet and even create your own components. VID is a dialect which allows the flexible implementation of complex graphical interfaces on multiple platforms. All this takes place in a single executable file that is only a few hundred KB in size.



```
Script requests permission to open a port for read/write on: /root/vw/public/www
.rebol.com/view/reb/demos.r
.bashrc COLORTERM= BROWS
ript-fu -gimp 8 7 64 64
.bashrc COLORTERM= BROWS
```

Figure 0-3. *Graphic programming with Rebol/View*

Rebol/View is also free and can be redistributed without constraint. A commercial version called Rebol/View/Pro is also available and adds to the evaluator a group of encryption functions (DES, RSA, DH and electronic signature management) and also support for the HTTPS protocol.

Rebol/Command

For *e-business* applications, you can use a commercial product called Rebol/Command. This version is also available for multiple operating systems and provides encryption functions, the HTTPS protocol, and access to ODBC data sources and the MySQL and Oracle databases.

Rebol/Command is perfectly adapted for the development of complex dynamic web sites and thanks to its native FastCGI support, it allows the establishment of high-performance applications that can be distributed across multiple servers using a n-tier architecture.

Rebol/SDK

Rebol/SDK is a development kit allowing the generation of executable files from a Rebol script. With the SDK, it becomes possible to distribute an application written in Rebol without the user having to install Rebol/Core, Rebol/View or Rebol/Command. One feature of the SDK is that it uses an encryption algorithm to conceal the source code of the Rebol script. Thus making it possible to distribute a commercial application containing proprietary technology.

Rebol/IOS

Finally, if you want to set up an intranet with a high degree of security and in which a community of users can share information, applications and data files, you can use the Rebol/IOS *groupware* platform. This is composed of two elements which are the server, Rebol/Serve, and the Rebol/Link multi-platform client. Rebol/IOS not only comes with a large number of applications ready for use (task management, phone book, news publication, administration tools, etc.) but also includes a development kit which allows custom applications to be written.

A network programming language

These different versions of Rebol support several TCP/IP protocols and make the development of applications able to exchange data over a network easy. Support for HTTP allows the development of web clients capable of surfing the World Wide Web or your own intranet in order to retrieve data. FTP offers the possibility to retrieve files from or send them to a remote server. The POP3 and SMTP protocols let you retrieve and send email. News forums can be accessed by using the NNTP protocol.

A script may also interrogate your DNS server to find the machines on your network or obtain information on user or a server with the help of the Finger and Whois protocols.

If these are not enough, you can also design your own protocols. In fact, Rebol provides direct access to TCP and UDP ports on your machine. It is possible to write not only client applications but also servers. A simple HTTP server can be implemented in Rebol in less than 50 lines of code!

Manipulating information

As well as its talents in the communications domain, Rebol also excels at managing information. It differs from other programming languages by the number and specialisation of its data types. Rebol is obviously capable of manipulating numbers, reals, strings or Boolean values but it doesn't stop there. Rebol is one of the few languages which naturally recognises and uses IP addresses and URLs.

These simple data types can be grouped together in lists. Lists are the foundation of the language. In Rebol, lists are omnipresent. A Rebol script is a list of words. An array is a list of values. Rebol provides the programmer with a set of words to access this information such as traversing a list and seeking or extracting data with the minimum of operations.

Object programming

Data can be contained in objects, grouping properties and methods. Objects written in Rebol allow the development of modular applications, facilitate team work on a project and enhance the productive aspects of the language. Conceptually much simpler than JavaBeans, Rebol objects can also be transmitted over a network, shared between multiple applications and have a mechanism for introspection.

The main applications written in Rebol

Many applications have been written in Rebol. Almost all of them are free and operate on all the platforms that have a Rebol evaluator. Their eclecticism shows the versatility of a simple, elegant language. Connect to the World Wide Reb and download protocols, dialects, web applications, utilities and even video games.

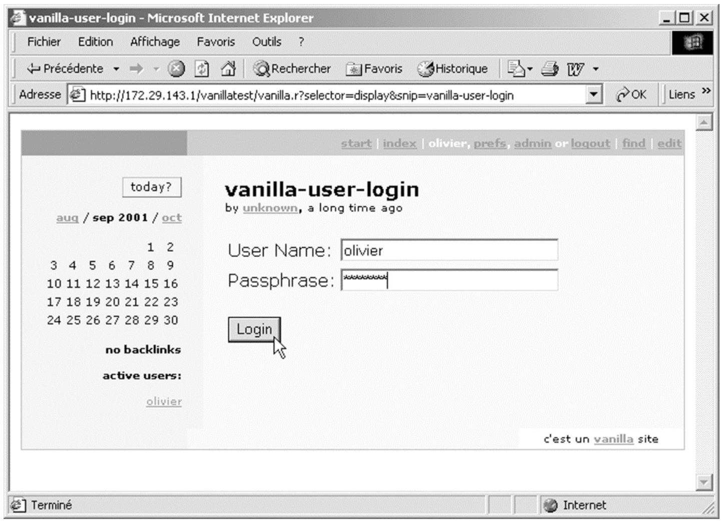
Protocols and dialects

Protocols and dialects enrich the capabilities of a Rebol evaluator. Protocols relate mainly to networking. Vincent Demongodin has written a SNMP (*Simple Network Management Protocol*) client for Rebol to collect information about the active elements on a network. Other essential products include the MySQL and PostgreSQL protocols of Nenad Rakocevic (www.softinnov.org). Free and compatible with all versions of Rebol, they allow you to interact with these famous SQL databases. There are in fact a number of additional network protocols that you can add to your Rebol evaluator.

Dialects are domain specific languages. Each of them is a language within a language and is dedicated to performing a task in the most efficient way. Gabriele Santilli’s PDF-Maker simplifies creating PDF documents. If you need to display numeric data as curves or histograms, grab the excellent Q-Plot which is a prodigiously effective dialect for such operations. In the Web domain, do not miss the SWF dialect which allows you to dynamically generate Flash animations.

Rebol and the Web

As Rebol is network oriented, it is hardly surprising to find many applications for the Web. MailReader and Rim are two communication tools, a mail client and a chat client. Rix is a search engine dedicated to documents about Rebol and Rebol source code. ShearchCenter allows the interrogation of multiple search engines. Vanilla is a web application for the establishment of collaborative sites.


vanilla-user-login - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente Rechercher Favoris Historique
Adresse
OK Liens
start | index | olivier, prefs, admin or logout | find | edit
today?
aug / sep 2001 / oct
Terminé
Internet

http://172.29.143.1/vanillatest/vanilla.r?selector=display&snip=vanilla-user-login

vanilla-user-login
by unknown, a long time ago

User Name: olivier
Passphrase: ********

Login

no backlinks
active users:
olivier

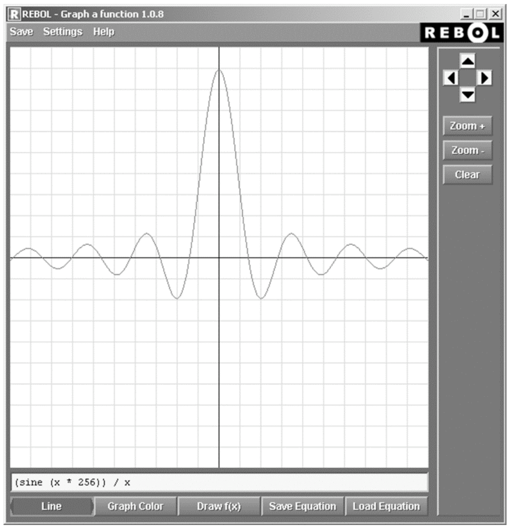
c'est un vanilla site

Figure 0-4. *The Vanilla wiki.*

Cheyenne is a very powerful HTTP server written entirely in Rebol. Impressive and hard to ignore, AltMe allows the creation of virtual communities.

Utilities as if it is raining!

Castro is a file manager allowing you to view, rename, move and delete files. With François Jouen's Michka, you can classify your PDF documents.



```
(sine (x * 256)) / x
```

Figure 0-5. *Draw your curves with Grapher.*

Essential for developers, Anamonitor fully monitors the system object of the Rebol evaluator. Finally Grapher will delight mathematics fans tracing and retracing all the curves imaginable.

Multimedia and games

Rebol is good for games. It's GCS graphics engine is totally independent from the execution platform and is capable of managing complex graphic operations. It can draw, apply effects (brightness, rotation, transparency, etc.) and even read images in many formats (PNG, GIF, JPG and BMP). The most impressive demonstration is most probably Reviewer which is a great product for storing and editing graphics.

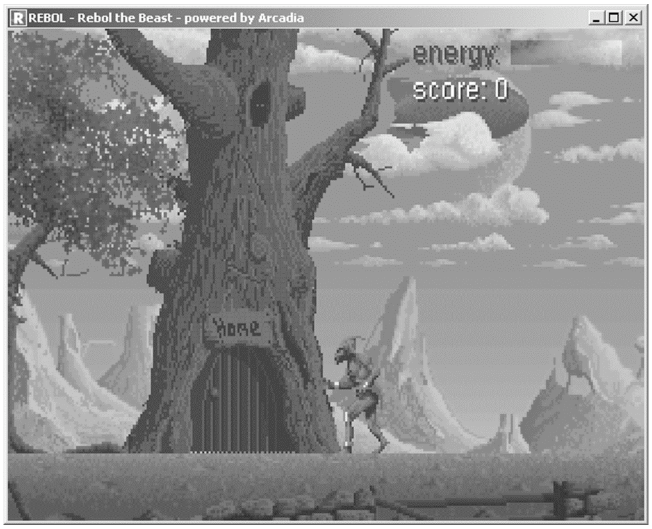


Figure 0-6. *A game made with the Arcadia engine.*

Rebol is lightweight, versatile and really simple to deploy. It has everything needed to develop games and multimedia applications. Rebol, a network oriented language with excellent graphic and animation ability, is a fantastic solution for creating online games.

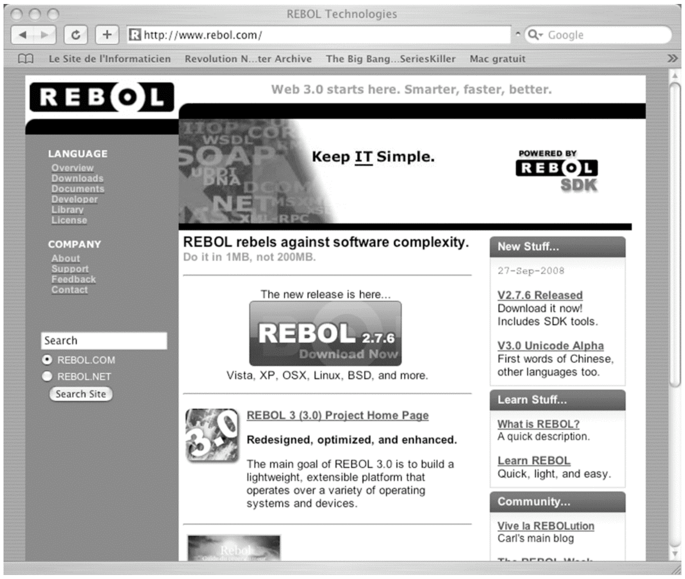
R-Box2 - Warehouseman's comeback!
Becherovka
Undo
New
Save
Load
Your warehouseman already did 32 steps (pushing the box 6-times)

Figure 0-7. *The R-Box2 game*

.
Talented programmers have already provided numerous demonstration of Rebol's online capabilites. Take a look at R-Box2 which is an adaptation of a classic video game. For his part, Cyphre has not hesitated to develop a video game engine called Arcadia, incorporating a variety of functions and a specialised dialect.

Downloading and installing

Rebol/Core can be downloaded from the *Download* section of www.rebol.com. Simply select the file archive for your operating system. The file obtained is in zip format under Microsoft Windows and tar.gz for Unix systems. On these, the shell command `tar xvzf` followed by the name of the archive allows you to decompress files into the current directory. There will be a number of files including the Rebol evaluator, installation documents, updates and Rebol scripts recognisable by their .r extension. On Unix-type systems, it is common practice to copy the Rebol executable to the /usr/local/bin directory so that it can be accessed by all users of the machine.



Web 3.0 starts here. Smarter, faster, better.

Keep IT Simple.

REBOL rebels against software complexity.
Do it in 1MB, not 200MB.

The new release is here...

Vista, XP, OSX, Linux, BSD, and more.

REBOL 3 (3.0) Project Home Page
Redesigned, optimized, and enhanced.
The main goal of REBOL 3.0 is to build a lightweight, extensible platform that operates over a variety of operating systems and devices.

27-Sep-2008
V2.7.6 Released
Download it now! Includes SDK tools.
V3.0 Unicode Alpha
First words of Chinese, other languages too.

What is REBOL?
A quick description.
Learn REBOL
Quick, light, and easy.

Vive la REBOLution
Carl's main blog

Figure 0-8. *The www.rebol.com site.*

Once you've done this, you must configure the evaluator to get the best use from it by specifying your email address, STMP server, POP server and any proxy server that you want to use. Use the setup.r script for this by typing the following command line `rebol setup.r`. This script is an interactive program in which you simply answer the various questions. The result is a `user.r` file, specific to each user. The folder also contains a file called `rebol.r`. Like the `user.r`, script this file is loaded by the Rebol interpreter whenever it is loaded. The `rebol.r` file is intended for storing features that will automatically be included in the evaluator. To discover all the words available in Rebol/Core, you can run the `words.r` script with the help of the command line syntax `rebol words.r`. Then you will get a file, `words.html`, containing all the definitions of each element of the Rebol dictionary.

The Rebol/View archive contains only three files and two of them are documentation. In fact, you simply launch the Rebol executable to trigger the installation procedure to set the directory to which Rebol/View will be copied and also indicate your email settings and possibly those of a proxy server.

Once the set-up is complete, the evaluator starts and displays a desktop that gives you access to different resources.

At the top of the window, the *User* menu item allows you to change your configuration. Through the *Goto* menu option, you can specify a URL to a Rebol script on the Internet and thus run it.

On the left, the *Rebol.com* folder gives you access to applications that are written in Rebol which are available over the Internet and can be downloaded over the Internet for running on your machine. Stored in a cache, these scripts are always available to you whether you are connected to the Internet or not. This illustrates one of the key Rebol concepts: the X-Internet. This idea describes an information exchange network in which each node is an active participant. Weighing less than 600 KB, with a network oriented scripting language, platform independent, economic with system resources and incorporating advanced graphic features, Rebol/View is the model client for such an architecture.

Below the *Rebol.com* folder, the *Public* folder gives access to the Rebol script library, hosted at Rebol.org, and the many RebSites hosted by various members of the Rebol community. These contain hundreds of useful scripts and, following the X-Internet principle, are also stored in a cache so that they can be used when you're not connected to the Internet.

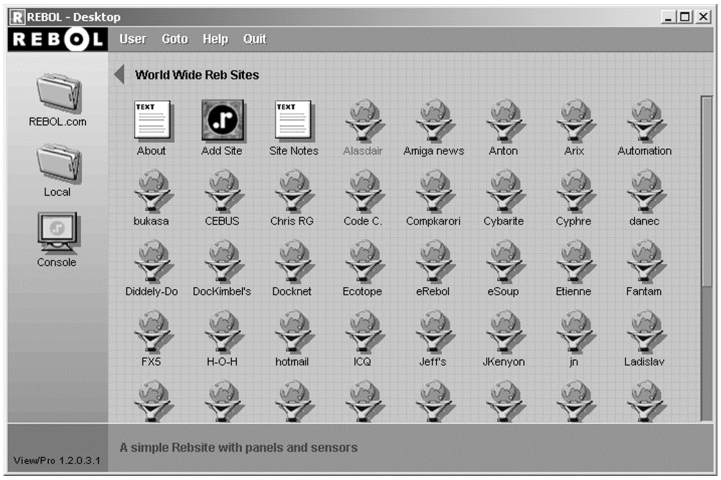


Figure 0-9. *Rebol/View provides access to RebSites.*

Next, you will see a *Local* folder for the classification and access to the different Rebol scripts on your machine. Finally, the *Console* option lets you launch an interactive console identical to that of Rebol/Core.

Using the console

The Rebol console holds many possibilities that facilitate working with Rebol. First, it allows you to work interactively and test code sequences. For example, you can type `2 + 2` followed by enter key. Immediately, the instructions are evaluated and the result displayed in the console.

The code sequences are not limited to a single line, it is possible to type full blocks before their evaluation. During typing, the right and left arrow keys move the cursor to allow you to make corrections. Using the up and down arrow keys, you can scroll through the previous commands and by hitting the enter key, re-run prior instructions. A single press of the tab key activates Rebol's auto-complete function. Pressing it twice in quick succession displays a list of the Rebol words that start with the characters already typed. For more information about a specific word, you can use the word `help` followed by the Rebol word about which you are enquiring. Each Rebol word is self-documented and you can do the same for your own functions. The `help` function displays the meaning of a word, its different parameters and its usage in the console.

There are different ways to run a script residing on your hard disk. You can specify the filename on the command line, create folders used locally by Rebol/View or use the word `do` followed by the filename. Thus to evaluate the `myscript.r` file, you type the command `do %myscript.r` in the console. The "%" character tells Rebol that the argument is of the file (`file!`) or path (`path!`) type. By extension, since Rebol is a network-oriented language, you can also execute a script stored on a remote server which will be read with the help of the HTTP or FTP TCP protocols. The instruction `do http://myserver/scripts/transfer.r` launches an application stored in the file system of a web server.

To help you develop your own scripts, the console includes some dedicated tools such as the word `trace` which takes a Boolean value (on or off) to enable or disable the monitoring of code as it runs. This instruction also lets you monitor network actions and reports the different function calls. Through the word `echo`, you can specify the name of a log file in which the information displayed in the console will be saved. When the word receives an argument with the value `none`, the logging is halted. `recycle` gives control over the Rebol garbage collector. Automatic memory management can be enabled or disabled. A special "torture" mode allows testing of extremes.

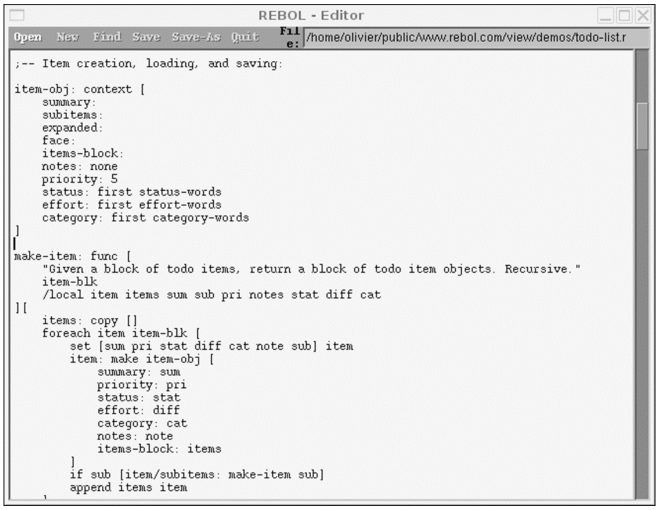
Establishing a working environment

The console is a very handy tool but is not sufficient to write Rebol applications. You will, of course, need a code editor to enter your scripts. In this domain, there are many to choose from and it is mainly a question of taste. There are configuration files for many editors which provide colour coding of Rebol code.

You can opt for products such as Vim (www.eandem.co.uk/mrw/vim/syntax) or Emacs (www.rebol.com/tools/rebol.el) for Unix.

Under KDE, the Kate editor can also be configured to recognise a Rebol script (www.errru.net/rebol/utilities/katesyntax). On MacOS X, the excellent editor SubEthaEdit (www.codingmonkeys.de) includes a syntax colouring file. On Windows, you find the remarkable and free Crimson Editor (www.crimsoneditor.com). If you regularly swap between different machines, you could try the free JEdit (www.jedit.org) which runs on any machine with Java installed and includes not only Rebol code colouring but also a nice bracket auto-alignment feature.

If you use Rebol/View, then you could use its integral editor, you can activate it by right-clicking on a file icon on the View desktop and pressing the *Edit* button or typing `editor` followed by the name of the file you wish to edit in the Rebol/View console.



/home/olivier/public/www.rebol.com/view/demos/todo-list.r

```
;-- Item creation, loading, and saving:

item-obj: context [
    summary:
    subitems:
    expanded:
    face:
    items-block:
    notes: none
    priority: 5
    status: first status-words
    effort: first effort-words
    category: first category-words
]

make-item: func [
    "Given a block of todo items, return a block of todo item objects. Recursive."
    item-blk
    /local item items sum sub pri notes stat diff cat
][
    items: copy []
    foreach item item-blk [
        set [sum pri stat diff cat note sub] item
        item: make item-obj [
            summary: sum
            priority: pri
            status: stat
            effort: diff
            category: cat
            notes: note
            items-block: items
        ]
        if sub [item/subitems: make-item sub]
        append items item
```

Figure 0-10. *The Rebol/View code editor.*

We have now reached the end of this introduction to Rebol. It has all been very theoretical so far but from here on we will start programming, even in the very first chapter.

1 Discover Rebol in an hour

For your Rebol apprenticeship, you are going to use Rebol/Core or Rebol/View which are freely available for all the principal development platforms. Whether you are using Mac OS X, Linux or Windows, you can start to learn this fantastic programming language.

The robotFTP project

By the end of this introduction, you will have developed a complete program and seen many different aspects of Rebol. You will now start a short tutorial to discover the main aspects of programming in Rebol. For this, the best way is to carry out a real project.

Project introduction

The objective is to develop an automated FTP client. This means you are going to write a script able to retrieve files from and send files to an anonymous FTP server. The different operations will be described in a configuration file. This script will allow you to deploy files according to a schedule or make a backup of files stored on a remote server. Used in conjunction with the Unix *cron* command, your script can be run periodically without any human intervention.

Technical considerations

In fact, you are going to write an interpreter whose task will be to implement a set of controls similar to those present in a conventional FTP client. The application can navigate through the tree structure of either the server or the client, send or receive a file, treat batches of files by their extension, create a directory, or delete a file or directory on the remote machine. To describe its operations, you will use a Rebol object containing the name of the remote host, the login and the password to be used, a boolean value to indicate a passive mode FTP connection and finally a list of the FTP commands to be executed.

With Rebol/Core, the script runs "silently" (needs no keyboard input or produces any screen output) and can be fully automated. However, if the script is run under REBOL/View, a graphical display will be used to show the progress of operations. A name is needed for any project and for this one it will be robotFTP.

The execution environment

First create a file called `robotftp.r` with a text editor.

For this file to be executable on Unix (and Mac OS X), you must specify the path to the Rebol executable on the first line of the file in order to invoke the script interpreter. So you should use the character sequence #! followed by the path of the Rebol executable. This is known as the shebang line.

It is often useful to provide Rebol with some additional information to define the runtime environment. The list of options can be obtained by running Rebol with the `–help` option. The `-c` option is used for running CGI scripts. If you want to directly evaluate a Rebol expression from the command line, you can use the `--do` option followed by the Rebol instructions. So, the command `rebol --do "print 2 + 2"` will display the result of the evaluation. This option is useful when you want to set the value of a variable before running a script.

Rebol scripts running under Windows don't need a Shebang line. You can simply double-click on the script and it will automatically launch Rebol (provided that the name of the file ends in .r and the r file extension has been associated with Rebol). When you run Rebol scripts this way, it is not possible to supply the additional information to define the runtime environment for each script. To do that you need to run Rebol from the Windows Command Prompt; for example `c:Rebol\Rebol.exe –c robotftp.r`.

For our robotFTP project, you are going to use the most common options which are `-q` for starting in silent mode (without any information displayed) and `-s` to disable Rebol's built-in security manager to avoid the interpreter requesting confirmation from the user for each resource accessed. Rebol's security manager allows precise control over the freedom of action of a script. This is very important with a network-oriented programming language with which it is possible to download and run applications via the Internet. The security manager allows monitoring of file system access as well as access to outside networks.

The security manager

The word `secure` specifies the behaviour of the interpreter in response to the script's requirements for file access (`read`, `write`, `execute` and `all`) establishing the rules based on four options (`allow`, `ask`, `throw` and `quit`). The logic of the security manager is to allow everything that is explicitly permitted and even allow the modification of the security policy if it is reinforced by the new rule.

Otherwise, the security manager is forced to ask the user for specific permission before allowing any file to be accessed. To avoid the redefinition of certain critical functions by a script, the security manager also provides the user with the `protect-system` word. As you can see security is not dealt with lightly in Rebol, which provides an extremely safe runtime environment.

For our robotFTP project, the first line of the script looks like this: `#!/usr/bin/rebol -qs` (obviously the file path depends on your installation).

Writing the header

The next step is to draft the header. Every Rebol script starts with an information section whose objective is to document the code. Within a Rebol data block, the programmer has the opportunity to use different fields to present the script. These metadata are directly readable by a human. The choice of fields is left completely to the author. However, it is wise to follow the standards established in the Rebol language documentation.

Chapter five of "*Rebol/Core Users Guide*" presents some rules to follow to write clear and legible Rebol scripts and also suggests standard fields to use for header information. By following this standard, it becomes possible to use the reflective features of Rebol and perform searches by author, category, date or version number of the Rebol scripts on your hard drive.

The main fields recommended in the documentation are `title` for the script title, `date` for the date written and `author` for the name of the programmer. You can also specify the name of the script (`name`), the filename (`file`), a version number (`version`) or the version of Rebol required to run the script (`needs`). With the fields `purpose`, `note` and `history`, the programmer can provide a detailed description of the purpose of the script, give an indication of its functionality and show how it has evolved.

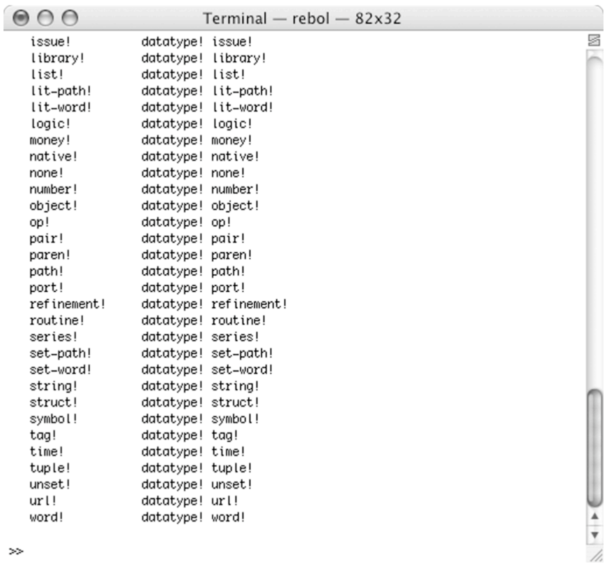
For the robotFTP project, you will learn the key fields and present the use of your FTP client. The header is prefaced with the word REBOL and the metadata are stored in a block, a fundamental data format in Rebol. As you can see, the allocation of a value to a variable is achieved by using the “:” operator.

```
REBOL [
  title: "automatic FTP client"
  author: "Olivier Auverlot"
  version: 1.0
  purpose: {
 RobotFTP executes of FTP commands
    in order to automate the sending and
    receipt of files
   }

  usage:
    robotftp commands-script
   }
 ]
```

The principle datatypes

The construction of the header is an opportunity to discover the many datatypes available in Rebol. Along with simple types such as `integer!`, `string!`, `logic!` or `char!`, the language provides types adapted to storing and processing large volumes of information with `block!`, `hash!` and `binary!`.

Rebol demonstrates its adaptation to network programming and information exchange with a number of dedicated types such as `tag!`, `url!`, `tuple!` and `email!`. In fact, you have more than fifty different datatypes at your disposal. All you have to do to get a list of them is to type the command `help datatype!` in the Rebol console.



```
issue!          datatype! issue!
library!        datatype! library!
list!           datatype! list!
lit-path!       datatype! lit-path!
lit-word!       datatype! lit-word!
logic!          datatype! logic!
money!          datatype! money!
native!         datatype! native!
none!           datatype! none!
number!         datatype! number!
object!         datatype! object!
op!             datatype! op!
pair!           datatype! pair!
paren!          datatype! paren!
path!           datatype! path!
port!           datatype! port!
refinement!     datatype! refinement!
routine!        datatype! routine!
series!         datatype! series!
set-path!       datatype! set-path!
set-word!       datatype! set-word!
string!         datatype! string!
struct!         datatype! struct!
symbol!         datatype! symbol!
tag!            datatype! tag!
time!           datatype! time!
tuple!          datatype! tuple!
unset!          datatype! unset!
url!            datatype! url!
word!           datatype! word!

>>
```

Figure 1-1. *Display of Rebol’s datatypes in the console.*

You should not be misled by this profusion of types; Rebol is not a strongly typed language (in the traditional sense) and places priority on flexibility. Even though you may declare a declaration with a given type, it may change at any time during the execution of a script.

Bear in mind that Rebol is able to determine the type of data. So, if you type the instruction `type? olivier.auverlot@domaine.fr`, in the Rebol console, the interpreter will unequivocally return the answer: `email!`.

And now, the code!

The first real assignment is reading parameters from the command line. In effect, when the `robotftp.r` script is launched it takes the name of a file containing the various FTP commands to be executed as an argument. For this, we will use the `args` property of the `options` object contained in Rebol’s `system` object. This latter is the core of the interpreter and is a tree-structure composed of a number of objects containing multiple operating parameters such as the type and version of the interpreter, the installation directory, the user’s directory, network protocols and statistics collected.
It also contains the whole Rebol dictionary. This unique store contains constants, functions written in Rebol and native functions written in C.

If you want to embark on a study of this object, all you need to do is to enter `print mold system` in the Rebol console.

Accessing a property or a method of an object is achieved with the help of the / character which sets an access path. To read the `args` property, you must write `system/options/args`. This property contains a block of character strings. You must therefore obtain the first element of the property and convert it to a file type before you load the object describing the FTP operations into memory and evaluate it. In Rebol, this suite of operations is performed in a single line command: `do load to-file first system/options/args`. To avoid an execution error if the list is empty or the file does not exist, this line should be enclosed in an "error-resistant" block preceded by the word `try`. The block is always evaluated. If an object of the type `error!` is returned, the second block is executed to manage the problem.

```
if error? try [
 commands: do load to-file first system/options/args
] [ error/fatal "Command script could not be loaded" ]
```

Declaring a function

If a problem is encountered, the error function is called. Its role is to display a message for the attention of the user and, if necessary, stop execution of the script. That decision is indicated by using a feature of REBOL known by the term *refinement*. This mechanism allows the modification of the behaviour of a function and the use of a variable number of parameters. In this case, it's the *refinement* `/fatal` which is used and it is declared within the function.

```
error: func [ "Display an error message"
 msg [ string! ] "Description of the problem"
 /fatal "The error requires stopping the program"
] [
 print msg
 if fatal [ quit ]
]
```

A function is a word of the `function!` datatype. It can be declared in different ways by using the `make function!`, `func` or `does` syntax.

The `error` word takes a character string called `msg` as a parameter and provides a *refinement* `/fatal`. Again Rebol allows the programmer to include metadata in the code. It is possible to indicate the purpose of the function and to document the parameters and *refinements* supported. Once a function has been evaluated, the `help` command extracts this information and displays it in the Rebol console. The body of the function simply displays the message supplied as a parameter and performs a simple test to determine whether the `/fatal` *refinement* has been used.

Manipulating URLs

Once the contents of the command line have been analysed, you can construct a URL with which to connect to the FTP server. In the object describing the operations, the `user` and `password` properties contain the account and password of the user. If the `user` property isn't present, the connection will be anonymous. You must therefore access the `user` property of the `commands` object and check that its value isn't `none`. You will notice in the code that `user` is preceded by a ' character. This tells the Rebol interpreter that this data item is a symbol and should not be evaluated by the interpreter.

The server name is indicated by using the `host` Property. As all the data has already been loaded into the `commands` object, all you need to do is to use the Rebol string manipulation functions to assemble the URL. In Rebol, a string is a list of characters that can be manipulated and traversed in the same ways as blocks. The dictionary contains numerous functions for copying, extracting, concatenating and searching for character sequences. The initialisation of the network parameters ends with the activation of passive mode by changing the Boolean value of the `system/schemes/ftp/passive` property. Finally, the script assigns the current directory tree of the FTP server to the curdir word. When the user moves through the folders, this variable will be responsible for storing the location.

```
ftpurl: copy "ftp://"
if not none? get in commands 'user [
 ftpurl: join ftpurl [ commands/user ":" commands/password "@" ]
]
ftpurl: join ftpurl commands/host

if not none? get in commands 'passive [
 if commands/passive = true [ system/schemes/ftp/passive: true ]
]
curdir: copy %/
```

Executing FTP commands

Now you need to define how the various FTP commands will be described in the configuration files. These are objects made up of various properties that indicate the connection settings (`host`, `user`, `password` and `passive`) and operations to be performed. For this, you use a property called `script` whose content is a block. This will allow you to use one of the most interesting features of Rebol: dialects.

Defining and using dialects

In effect, Rebol can define unique languages whose function is dedicated to a specific task. These are specialised vocabularies that are not part of the Rebol dictionary but which can be defined and manipulated by the user. Using a grammatical analyser following the BNF (*Backus-Naur Form*) notation, it is thus possible to define mini-languages in Rebol.

The dialects can be used in many areas such as describing operations, establishing a network protocol, or defining constraints and rules. Moreover, the Rebol interpreter itself contains different dialects dedicated to the manipulation of strings or the description of graphical interfaces. Several libraries also use this additional functionality to elegantly extend the capabilities of the language, so you can find on the Internet a dialect to generate PDF documents (www.rebol.org) or yet another which simplifies managing consoles in text mode (http://www.rebolforces.com/articles/tui-dialect).

The robotFTP dialect

In the case of the robotFTP project, you need a dialect whose syntax is similar to the commands for FTP clients. So it is a set of commands, each of which may have an optional parameter. The definition of the rules takes place in a block where each symbol relates to a possible command. If the instruction receives a parameter, you must define the type of this parameter and the name of a variable that will receive its value. The rest of the definition is enclosed in parentheses and contains Rebol code to be executed when the command is encountered during the analysis.

In the following code, you will probably notice that the definition of the different commands is enclosed in an `any [ ]` block and each line is terminated by the "|" character. The reason is quite simple: this is to indicate to the interpreter that at least one of the grammatical conditions must be met for the analysis to be correct (the "|" is the syntax for "or").

```
rules: [
 any [
  'quit (quit) |
  'debug set arg string! (print arg) |
  'lcd set arg file! (exec-cmd [ change-dir arg ]) |
  'get set arg file! (exec-cmd [ write arg (read/binary make-ftpurl arg)
]) |
  'put set arg file! (exec-cmd [ write/binary (make-ftpurl arg)
read/binary arg ]) |
  'cd set arg file! (
   either arg = %/ [
    curdir: copy %/
   ] [ append curdir reduce [ arg "/" ] ]
  ) |
  'mkdir set arg file! (exec-cmd [ make-dir make-ftpurl/directory arg ])
|
  'delete set arg file! (exec-cmd [ delete make-ftpurl arg ]) |
  'rmdir set arg file! (exec-cmd [ delete make-ftpurl/directory arg ]) |
  'mput set arg file! (
   exec-cmd [
    foreach file read %. [
     if (suffix? file) = arg [
      write/binary (make-ftpurl file) read/binary file
     ]
    ]
   ]
  ) |
  'mget set arg file! (
   exec-cmd [
    files: read make-ftpurl ""
    forall files [
     files: first files
     if (suffix? files) = arg [
      write/binary files read/binary (make-ftpurl files)
     ]
    ]
   ]
  )
 ]
]
```

Managing errors

Some of the commands are very simple. The `quit` instruction simply closes the FTP connection and ends the Rebol interpreter session. The `debug` command receives a character string as a parameter which it displays on the screen. Other instructions are a little more complicated as they are designed to use the FTP protocol. For this reason, the execution of code is delegated to a function, `exec-cmd,` the purpose of which is to manage errors.

```
exec-cmd: func [
 cmd [ block! ] "Rebol Instructions"
 /local err
] [ if error? err: try [ do cmd ] [ print mold disarm err ] ]
```

This function takes a block containing Rebol code as its parameter and defines a local variable called `err`. This is used to gather error objects intercepted by the `try` word. The Rebol instructions are executed by using `do` which simply evaluates a block of code.

In case there is a problem, `disarm` transforms the datatype `error!` into an object which is then displayed on the screen. This object contains various information including a plaintext message describing the error.

The word `mold` is designed to prepare information of any datatype for displaying on the screen. For example, a character string will be enclosed in quotes, a block in square brackets. In this case, it will be an object that will be displayed.

Conditional expressions

The navigation around the various directories within the FTP server is entrusted to the `cd` command. Its work is to update the contents of the variable, `curdir`, by the use of a simple test. For this, Rebol offers two words; `if` and `either`.

`if` is followed by a condition and a block of code to be evaluated if the condition is true. `either`, for its part, is followed by a condition and two blocks of code: the first is evaluated if the condition is true, the second if the condition is false. With regard to operators, Rebol is very classical as you can use =, <, >, <=, >= or <>.

In the robotFTP script, if the argument provided is the value of the file system root (`%/`), this value is assigned to `curdir`. If not, the outcome of the evaluation of the block is that the argument and a "/" character are appended to the current directory.

Managing files

The other functions use words concerned with manipulating files. These commands interact with the server using the FTP protocol and therefore need to establish a url based on the argument passed as a parameter. For this, you are going to define the word `make-ftpurl` which returns a value of the `url!` datatype. As a first step, the argument of either the `file!` or `string!` datatype is "cleaned" by using the word `clean-path`. Thus, a path like `%/home/olivier/docs/../autres/` will be converted to `%/home/olivier/docs/`. You also need to distinguish between generating a URL to a file or a directory. For this, you must define a *refinement* `/directory` which, if used, will force a "/" character at the end of the url.

```
make-ftpurl: func [ "Constructs the URL for FTP commands"
 arg [ file! string! ] "Argument passed by the dialect"
 /directory "The URL is for a directory"
] [
 clean-path curdir
 either not directory [
  to-url join ftpurl [ curdir arg ]
 ] [ to-url join ftpurl [ curdir arg "/" ] ]
]
```

The `lcd` command can specify a local directory by using the `change-dir` word. The creation and deletion of directories on the FTP server are entrusted to the `mkdir` and `rmdir` commands which call the words `make-dir` and `delete` of the Rebol dictionary. The commands `put`, `get`, `mput` and `mget` are more complex as they read or write files, either on the client or the FTP server. For this, they use the Rebol words `read` and `write`, and also the `/binary` *refinement* to work in binary mode.

The command `mput` allows a group of files with a common extension to be sent to the FTP server. `read` returns a file list in a block. This list is then traversed using `foreach` and each value is assigned, in turn, to the variable `file`. With the word `suffix?,` its extension can then be compared with that specified in the argument. If the boolean `true` is obtained, the file is transferred to the server. The `mget` command performs the inverse operation but uses another word designed to trawl through lists: `forall`.

Completing and testing robotFTP

You can now complete your application by inserting the line of code `parse commands/script rules`.

This instruction will trigger the application of the parse rules on the content of the property `commands/script`. To test your application, use a text editor to write a little test script called `test.r`. This contains an object consisting of four properties for the connection parameters and the `script` describing the operations to be carried out.

```
context [
 host: "monserveur.domaine.fr"
 user: "olivier"
 password: "homer"
 passive: false

 script: [
  debug “debut”
  lcd %temp
  mget %.r
  get %gscite159.tgz
  mput %.c
  debug "fin"
 ]
]
```

You now need to run it by typing `./robotftp.r test.r` in the shell. In case of a problem, check that your script has execution privileges (`chmod +x`). (Under windows, you would type c:\<insert path to Rebol>\rebol robotftp.r test.r). If all went well, the operations described in the `test.r` file were performed. The program is well suited to run under a scheduler such as the Unix cron utility as it can run silently. In certain cases, a graphical display can also be interesting. Now you are going to modify the script to take advantage of the features of Rebol/View.

Adding a graphic layer

Our aim is to add graphic support to the script to demonstrate the capabilities of Rebol/View. We will open a window containing a progress bar and display the progress of the FTP commands.

The script will be run on different versions of Rebol, so it is necessary to determine which version of the interpreter is being used in the script. For this, you use the `view?` word which returns the value `true` if the script is being executed by Rebol/View.

Managing a progress bar

At the start of the script, you will add a line of code to initialise the variable `stp`. A progress bar works on a range of values from 0 to 1 (100%) and `stp` is the value that will be added to the progress bar on the completion of each FTP command. To calculate a reasonable step value, we divide the number 1 by the number of commands in the block `commands/script` (by approximation, the number of elements divided by two).

Opening a window

Next we define a function, `progress-window,` which opens a window in the centre of the screen. The window's layout (`layout`) is defined by using Rebol's VID dialect and is assigned to the object `ftp-box`. This includes a title (`title`) and a progress bar (`progress`) assigned to the object, `progression`.

```
if view? [
 stp: 1 / ((length? commands/script) / 2)
 progress-window: does [
  view/new center-face ftp-box: layout [
   title "robotFTP"
   progression: progress
  ]
 ]
]
```

Integration with the RobotFTP script

To update the progress bar automatically, it is necessary to modify the `exec-cmd` function so that it takes advantage of Rebol/View. For each FTP command executed, the `data` property of the `progression` object is incremented and the graphic component refreshed.

```
exec-cmd: func [
 cmd [ block! ] "Instructions Rebol"
 /local err
] [
 if error? err: try [
  do cmd
  if view? [
   progression/data: progression/data + stp
   show progression
  ]
 ] [ print mold disarm err ]
]
```

Before starting the interpretation of the rules, it is now necessary to call the `progress-window` function so that the window is displayed. Once all the commands have finished, a modal confirmation dialog box is displayed before the main window is closed.

```
if view? [
 request/ok "Operations completed"
 unview ftp-box
]
```

Summary

You are at the end of this short exploration of the Rebol language in which you covered a number of topics such as writing a script, network programming and developing a graphic interface. This small project allowed you a glimpse of the many facets of a novel language, resolutely different and designed to encourage the interchange of data.

2 The Rebol language

Let’s get started! Type the following instruction in your Rebol console:

```
print "Cuckoo"
```

Press the "Enter" key on your keyboard and you will immediately see the character string "Cuckoo" on your screen. Congratulations we have just performed our first evaluation. We used the word print whose function is to display the parameter that immediately follows it on the screen. Don’t worry about capital and ordinary letters, Rebol doesn’t care: it is not at all case sensitive.

Survival Guide

Let’s continue our exploration by looking at the word `what` : it displays the whole of Rebol’s dictionary in the console. If a Rebol word interests you, Rebol can provide you with information about the role and function of the word.

If you want to know what the word `input` does, simply type the phrase `help input` and you will get a description on your screen. Even better, type `source input`, you can now look at the source code of the word `input`. This operation is possible for all the words in the dictionary that are defined in Rebol itself. The Rebol interpreter makes the distinction between words written in Rebol (*mezzanine functions*) and those created in native code (*native functions*) for which the source is not available.

Let's write a program

Until now we have only directly evaluated our words in the Rebol console. We haven't written a program yet. We can write a Rebol program using a simple text editor. All software written in Rebol must start with a descriptive block allowing you to document your code. This block can be empty but it must always be present

```
Rebol [
      Subject: "my first program"
      Author: "Olivier Auverlot"
      Version: 1.0
]
```

You can even create your own headings according to you needs (update descriptions, objects or libraries required, date of last modification, etc…). There is a recommended set of headings but you are not forced to use them. This freedom allows you to adopt the descriptive block to the needs of each project; the important thing is to make sure that there is consistency between different software components. So our block could also be written:

```
Rebol [
      Title: "my first program"
      Author: "Olivier Auverlot"
      Version: 1.0
      Date-released: 10/11/2000
]
```

What will our first program to do? It will simply display a string of characters on the screen, wait until the "Enter" key is pressed and then stop running. To do that, add the following code immediately after your descriptive block:

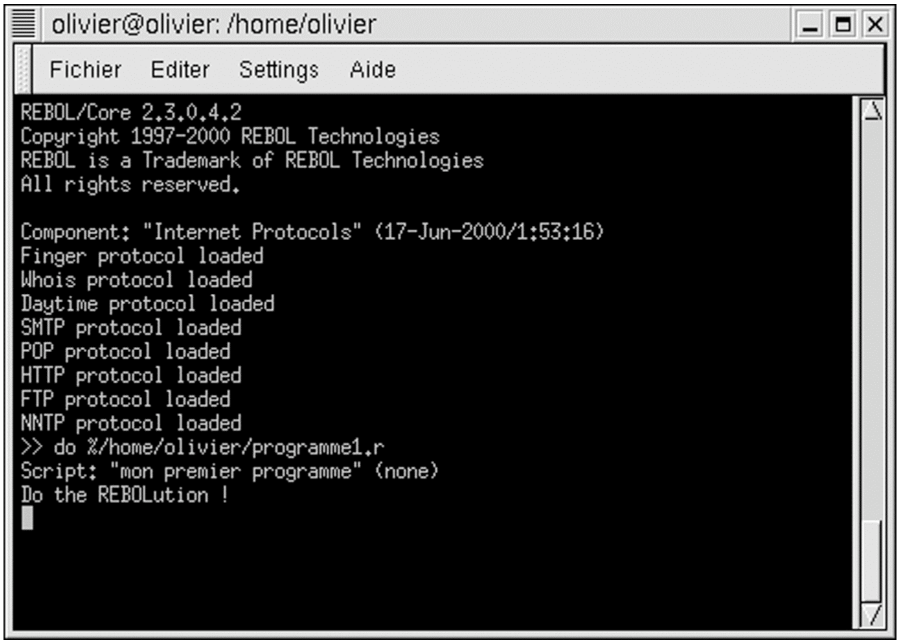
```
print "Do the REBOLution !"
input
quit
```

Running Your Script

Save this program on your disk with the name `programme1.r`. If you are working with Windows, all you need to do is double-click on the icon of your script's file. Under Linux and Mac OSX, you can run the script by entering the shell command `rebol programme1.r`

A second option under Linux and Mac OSX is to link the script to the Rebol interpreter. First insert the instruction `#!/usr/bin/rebol –qs` as the first line in your script; make sure it is before the opening Rebol block. (This line is called the shebang line in the Unix world). As you can see, we can pass runtime options to the Rebol interpreter as it is launched. When our script is run, the Rebol interpreter will not display any messages or impose any security restrictions. You can get a full list of the available runtime options by entering the word `usage` in the Rebol console. Then once you have added the shebang line and saved the file again, you must then make the script executable by entering the shell command `chmod +x programme1.r`.

You also have the option of running the script directly from the Rebol console. The `do` word can be used to run a script. It takes the name of the script to execute as a parameter; you can specify an access path to the script if necessary. In Rebol, a file is a datatype recognised by the interpreter because it starts with the "%" character. For this reason, if you want to run our script and it is stored in the directory `/home/olivier`, you type `do %/home/olivier/programme1.r` in the Rebol console.



```
REBOL/Core 2.3.0.4.2
Copyright 1997-2000 REBOL Technologies
REBOL is a Trademark of REBOL Technologies
All rights reserved.

Component: "Internet Protocols" (17-Jun-2000/1:53:16)
Finger protocol loaded
Whois protocol loaded
Daytime protocol loaded
SMTP protocol loaded
POP protocol loaded
HTTP protocol loaded
FTP protocol loaded
NNTP protocol loaded
>> do %/home/olivier/programme1.r
Script: "mon premier programme" (none)
Do the REBOLution !
```

Figure 2-1. *Running a script in the console.*

Variables and datatypes

After seeing how to use the console and the structure of a Rebol script, we will now create our first words. Rebol programming consists of expanding a dictionary of words which represent functions, object and variables.

Declaring a variable

It is extremely simple to create a word in Rebol. All you need to do is add the character ":" at the end and indicate its value. So the expression `var: 0` adds a word called `var` with the value number 0 to the dictionary. With the Rebol language being case insensitive, we could have also used `Var`, `VaR`, etc. If we then enter the word in the console, we obtain the value 0 as Rebol evaluated the content of the word which now forms part of the dictionary in its global context. However, if you use the word `what`, the word `var` seems to be missing from the dictionary. The answer to this question is obvious: `what` does not display variables. There is a way to check that a variable was actually declared; use the Rebol help function: `help var` (or its shorthand? `var`).

This not only confirms that the word has been created but also shows that the type of its value is an integer and its value (0). Nothing surprises you? For what reason did Rebol declare our variable completely when we didn't specify a type? The explanation is that in absence of a specific declaration, the interpreter selects the datatype that seems appropriate.

Another concept can and may possibly shock many purists: a word does not have a fixed type; it assumes the type of its contents and may well change during the course of execution. This means our variable may well be an integer when a script starts to run, then change to a character string halfway through and finally end up as a real (floating point) number. All these transformations have an element of danger but they are all possible. You will see later on how useful this potential danger can be once you've learnt how to tame it.

Finding the type of a variable

If during the execution of a script we want to find out the type of a variable's contents, we use the word `type?` Entering `type? var` in the console gives us `integer!`, which makes perfect sense. Now let us try to change the content of `var` by keying in `var: 0.82` and checking its type in the same way. Now we get `decimal!`. The type of our word has now properly changed in-line with the change of its content from a whole number to a floating point number.

Using Constructors

Using Rebol's constructors allows you to specify the datatype of a word. You no doubt can see that you don't have to use them but their use is strongly advised to help understand and maintain long scripts. Suppose we wanted to define the type of our word `var` as `date!`, the syntax to do so is `var: make date! 1/1/2000`. Don't be surprised by the direction of this expression, the word `var` definitely will have the type `date!` and is initialised to the 1st January 2000. Also never forget that its type can change constantly during execution.

A practical use of constructors is in converting types. For example, `var: make integer! 3.2` gives us the way to extract the whole part of a number (the word `var` will contain the value 3 and be of type `integer!`).

Simple datatypes

In this category, we will find the principal datatypes present in any programming language. We've already met `integer!`, `decimal!` and `date!`. Rebol also includes the types `time!` for the time, `money!` for amounts of money, `logic!` for boolean (`true` or `false`, `on` or `off`, `yes` or `no`), `char!` for single characters and `none!` which indicates there is no value present. The following examples show these different types:

```
int: make integer! 0
float: make decimal! 2.98
hour: make time! 15:35:00
date: make date! 1/1/2000
cash: make money! $10
boole: make logic! true
character: make char! #"A"
undefined: make none!
```

Complex datatypes

Rebol's complex datatypes are series which are made from simple types. These are the tools which give Rebol its power while making it possible for the programmer to easily handle data storage blocks, access paths to the files, URL or electronic mail addresses. Rebol is truly a language adapted to the daily needs of developers of client/server, n-tier and web applications. There are over fifteen complex datatypes in Rebol which will surely provide for your every need.

Character strings have the `string!` datatype. Strings are classically contained between double-quotes except when the string includes carriage-returns or double-quotes, then you must enclose the string in curly-brackets `{}`.

This ability to cope with those special characters is really appreciated by web programmers who can directly insert their HTML in Rebol scripts:

```
codehtml: {
      <html>
      <head></head>
      <body>
      My HTML page.
      </body>
      </html>
}
```

For binary data, we have the astute `binary!` type that allows us to specify in which base our data is stored. Suppose we want to assign a byte in base 2 to our word, we would use the declaration `data: make binary! 2#{01111111}`, which is the value 127.

Filenames and access paths are represented by the `file!` type. They always begin with the `% character`. If we enter `f: %file.txt` in the console, the expression `type? f` tells us that `f` has the `file!` type. We can always convert a character string to a filename by using a constructor. The syntax to use is `f: make file! “file.txt”`.

For the net, both the Internet and Intranets, Rebol has a veritable host of types. The declaration of an email address is simply `my-email-address: make email! olivier@domaine.fr`. Access paths to web resources (Uniform Resource Locators) are, as you will probably have guessed, represented by the `url!` type. The expression `web: make url! http://site.domaine.org` is how we assign an address to a word. TCP/IP do-it-yourselfers will appreciate the presence of the `tuple!` and `port!` types which represent an IP address and a socket respectively.

Blocks

Data blocks are also a complex datatype and are one of Rebol’s key concepts. They are at the heart of all components of the language. In Rebol, everything is a block. A script header is one, data and instructions are grouped in them, blocks are contained within blocks, and a script is a group of blocks. For Rebol programmers, the world is a block !!!

More seriously, blocks contain the structures in which to store information enclosed between square brackets i.e. `[` and `]`. There are a few different constructors with which to define blocks. The main one is `block!` and it works with a list of values of differing types organised without any restrictions or fixed-formats. So we can mix the various Rebol datatypes and, of course, include other blocks as well. The following example is completely valid :

```
myblock: [
      "abcd"
      olivier@domaine.fr
      [ 1 2 3 ]
      %file.txt
      [ 255.255.255.0 [ 80 129 ] ]
]
```

By default, Rebol gives the `block!` type to a list of values if no type is specified. For large, frequently viewed blocks of information, Rebol also provides the `hash!` type which allows the optimisation of data searches.

Handling lists

Hopefully, you are rapidly coming to understand that lists are fundamental in Rebol as they make it possible to store and manipulate data. A good understanding of them is essential to be able to use the language correctly.

Arrays

It probably won't surprise you to find that in Rebol, an array is a list. The word `array` allows the definition of an array with n elements. Enter `mytab: array 5` into the Rebol console.

You have just defined an array of five elements; each initialised to the value `none`. If you want to initialise each element of the array with the integer 0, you can use the *refinement* `/initial` :

```
myarray: array/initial 5 0
```

Refinements provide options for words. In fact, they allow a variable number of parameters to be passed to a word. Additional parameters are written in the same sequence as the refinements which allow you to use them. We will see later how to define refinements in our own words.

It is also possible to create an array with multiple dimensions by using a list with the word `array`. Suppose we wanted a two-by-five array, you only have to type:

```
myarray: array [ 2 5 ]
```

Arrays in Rebol are just lists, both are handled with the exactly the same words. Before we continue, I would like you to create the following list in the Rebol console:

```
colours: [ 1 "red" 2 "green" 3 "blue" ]
```

Navigating within a series

A list may be considered as a database. A pointer is used to traverse the different elements of a list. It marks the starting point from which data can be read. The words `head` and `tail`, respectively, let you place it at the beginning or end of a series, `next` and `back` let you advance or move back within the series. To find out if we are at the beginning or end of the series, we can use the words `head?` and `tail?` which return a boolean value. The following example places the pointer at the start of the colour series moves forward one element and checks to see if we have found the last element.

```
colours: head colours
colours: next colours
tail? colours
```

The word `index?` lets us find out at which element the pointer is positioned. It is also possible to place it directly at a given position with the word `at`. The new position is relative to the existing position of the pointer in the series. If we now enter `colours: at colours 1,` we don't get the value 1 but the character string "red" which is the element with which the pointer was aligned at the time we entered the phrase.

In the same way `length?` returns the length of the list from the position recorded in the pointer. (Incidentally, the length of a list in Rebol is simply a count of the number of elements in the list).

Accessing an element

Accessing an element in a list is very intuitive. First let's return our pointer to the first element in the list with `colours: head colours`. We can now use the words `first, second, third, fourth`, `fifth, sixth, seventh, eighth, ninth` and `tenth`. If we want the second element of the list, simply enter `second colours`. This method is perfect for the first ten elements of the list but what happens if our list has more than ten? We have the option of retrieving an element at a specific position in the list. There are two alternatives for this: `colours/2` or `pick colours 2`. These two instructions will display the second element of the list `colours`.

Adding and removing elements

Adding an element or another list to an existing list can be done through either insertion or concatenation by using the words `insert` and `join`. Suppose we wanted to add the block `[ 4 "yellow" ]` to our list, we can choose between two methods: `colours: join colours [ 4 "yellow" ]` or `colours: insert tail colours [ 4 "yellow" ]`. The `join` word makes it possible to combine elements while in this case `insert` adds the block to the end of the `colours` block. That is because we move the pointer to the end of `colours` with the word `tail`. As you would expect, if we want to insert another block at the beginning of the `colours` list, all we have to do is `colours: insert head colours [ 0 "white" ]`.

The `remove` word allows you to remove the element at which the pointer is positioned from the list. If we want to remove the second element from the pointer, we simply type `remove at colours 2`.

Modifying a series

As well as `pick`, we have `poke` which will bring back fond memories for old Basic programmers. `poke` allows you to change a value in place within the series. Like `pick`, `poke` index 1 always points at the first element of the series from the current pointer. The expression `poke colours 2 "violet"` will replace the string "`white`" with the string "`violet`" (as long as you inserted `0` and “`white`” at the beginning of the list and the current pointer points to the head of the list).

The word `change` also has the same effect: `change at colours 2 "white"` puts the original value back in our list.

Searching and sorting series

Rebol has two words for searching for elements within a list. If the element isn’t found, these two words return the value `none`. `find` returns the rest of a list starting from the position of the element being searched for. If you enter `find colours "blue"`, the interpreter displays `[ “blue” 4 "yellow" ]`.

The `select` word behaves differently because it returns the element after the one being searched for. `select colours 4` gets us the value "yellow". `select` is perfect for look-ups in a configuration file while `find` is useful for searching a database held in memory.

We also have the ability to sort the contents of a list in ascending order by using the word `sort`. (Rebol even allows the programmer to replace the standard sorting algorithm with their own).

Copying and clearing series

The word `copy` lets you copy one series to another. Any existing data in the destination list is lost. The expression `mycolours: copy colours` makes a copy of the elements of `colours` in the list of `mycolours`. Why isn’t the operation "mycolours: colours" sufficient?

Perform a simple test by creating a list `a` containing `[ 1 2 3 4 ]`. Now assign `a` to `b` with the expression `b: a`. If you change the first value of `a` to `10` (`a/1: 10`) and then type `b` in the console to find its value, you will notice that the first element of `b` has also become 10. Why? The answer is simple: when assigning one list to another they, in fact, share the same space in memory. For this reason, modifying an entry in one also modifies it in the other. It is necessary to be careful when using such an approach and it is best to consider using the word `copy` which creates a real copy of one list in another (At this moment, I'd like to remind you that a character string is also a list!). In this case, the second list contains the same elements as that of the original one but has its own space in memory. The two are totally independent.

Clearing a series is done with the word `clear` followed by the name of the list. The word used for the name of the list is not removed from the dictionary but is emptied of its contents. If you wish to completely delete a word, you can use the word `unset`.

Many of the words which we have just seen have many refinements which make the life of the programmer much easier. It would take too long to go into them in all in detail here. I suggest that you make use of `help` in the console which allows you look into all the different options available. Try to fully understand how to work with lists; these words are truly the base of the language.

Control structures and loops

After having discovered some of Rebol's data handling capabilities, we will now look at the control structures and loops available in the language. What is surprising on first seeing them, is the sheer number of options available to the programmer. Rare are languages which offer such freedom.

Tests in Rebol

In programming, tests allow the modification of the execution path of a program according to their evaluation. The scheme is simple: if a condition is true [do something] if not [do something else].

Rebol adopts this very traditional model. The word assigned to this operation is `if`. If the evaluation of its first parameter returns a boolean `true`, the code placed in the second parameter will then be executed. Tests use the classic operators such as, <>, >, <, <=, >= and the equals sign also allows the comparison of character strings. Contrary to many other languages, you don't have to enclose your tests within parentheses but experience shows that doing so can help to increase program legibility. The following program displays "OK" when the user enters the number 0:

```
REBOL [ ]
nbr: to-integer ask "a number ?"
if ( nbr = 0 ) [ print "OK" ]
```

The words `all` and `any` provide a concise form of the logical operations AND and OR. With `all`, the test returns `true` if all of its conditions are true. On the other hand, `any` returns `true` if any of its conditions are true. The following example displays "ALL true" if both of the variables `a` and `b` are 0, it then displays "ANY true" when it finds that `c` is not 0 but that `a` is 0:

```
REBOL []
set [ a b c ] [ 0 0 1 ]
if all [ (a = 0) (b = 0) ] [ print "ALL true" ]
if any [ (a = 0 ) ( c = 0 ) ] [ print "ANY true" ]
```

(Did you notice that we used the word `set` to quickly create and initialise a list of variables? Why not look it up in the Rebol console to find out more about it.)

And else?

Up to now, we have seen what happens only when a test turns out to be "true". But generally, we must also manage the other case: when the result of the test is the boolean `false`. In Rebol, we have to ways to do so. First of all, we can use the *refinement* `else` available for the word `if`. The first block of code is executed if the condition returns true, if not the second block is evaluated. In the following code, we display true or false depending on whether the variable `a` contains the character string "Rebol" or not:

```
REBOL []
a: copy "Rebol"
if/else ( a = "Rebol" ) [ print "true" ] [ print "false" ]
```

We can also use the word `either` which is actually more in the spirit of the language. A little history, the refinement `else` of the word `if` had been requested by Rebol programmers on the mailing list. Originally, only `either` allowed a test of the form IF…THEN…ELSE. It is very simple and more stylish (though that is a matter of taste!):

```
REBOL []
a: copy "Rebol"
either ( a = "Rebol" ) [ print "true" ] [ print "false" ]
```

Multiple choices

With the word `switch`, you can selectively execute portions of code; the choice of which code to execute is made on the basis of the value of a variable. The `default` *refinement* allows you to handle cases when the value doesn't match one of those specified'

```
Rebol [ ]
choice: to-integer ask "What is your choice (1 – 3)?"
switch/default choice [
      1 [ print "choice 1" ]
      2 [ print "choice 2" ]
      3 [ print "choice 3" ]
] [ print "This choice was not expected" ]
```

Loop

The word `loop` repeats a sequence of code a specified number of times. (Of course, the specified number must be a positive integer.). `Loop`'s first parameter is the number of repetitions and the second a block containing the Rebol instructions to be repeated. Our example asks you the number of times the program should display "Hello" on the screen and then carries out the operation.

```
REBOL [ ]
nbr: to-integer ask "Number of repetitions ?"
loop nbr [ print "Hello" ]
```

The for loop family

`forever` is the simplest of this group of words. It simply executes the code passed to it as a parameter: `forever [ print "Stop me by pressing ESC !" ]`.

The word `for` runs a portion of code a certain number of times but this time the programmer can use a counter for the loop and control its incrementation. `for` is a word which requires no less than 5 parameters:

- A variable to act as a counter,
- The value of the counter at the start,
- The value of the counter at the finish,
- The amount the counter should be incremented on each iteration of the loop ,
- The Rebol code to be executed.

The following program inserts ten integers (1 to 10) in a list.

```
REBOL [ ]
list: copy []
for i 1 10 1 [ append list i ]
```

The words `forall`, `forskip` and `foreach` are extremely useful as they allow you to traverse the elements of a list. On each iteration of the loop, `forall` moves the pointer onto the next element in the list. It makes it really easy to display the contents of our list in the following way:

```
forall list [ print first list ]
```

`forskip` traverses a series while skipping over a specified number of elements until it reaches the end of the list. This displays every second element of our list:

```
forskip list 2 [ print first list ]
```

After `forall` and `forskip` have done their work, the list's pointer will be found at the end of the list. To perform more actions on the list, it is necessary to return the pointer to the start by using the word `head`.

`foreach` also traverses each element of a list but it assigns the value of a pointer to a variable:

```
foreach value list [ print value ]
```

Contrary to `forall` and `forskip`, using `foreach` does not require putting the list's pointer back at the start of the list. It stays in its original position.

Repeat, until and while

The word `repeat` makes it possible to repeat the evaluation of a block of code a specified number of times, a variable is used as a counter for the loop which is fixed to start at 1 and is incremented by one for each iteration of the loop. This variable does not belong to the global context and it doesn't exist outside the block of code being evaluated. The following example displays 1 2 3 in spite of `i` being initialised to 5 at the start of the program.

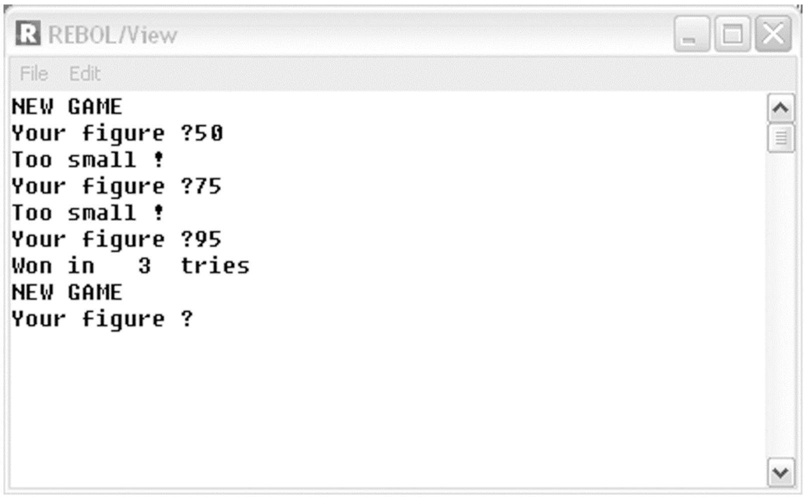
```
REBOL []
i: 5
repeat i 3 [ print i ]
print i
; the loop has finished, i is always worth 5 !
```

The two words `until` and `while` permit the execution of a program sequence an undefined number of times. The difference between the two words is that `until` evaluates the code until the last evaluation in the block returns the boolean `true` whereas `while` evaluates the code as long as the supplied condition is true. (Actually, `until` keeps evaluating the code as long as the last evaluation in the block returns `false` or `none`). The following example uses the two methods; the objective is to increment a numeric value until it reaches the number 10:

```
REBOL []
i: 0
until [
      i: i + 1
      ( i = 10 )
]
i: 0
while [ i < 10 ] [ i: i + 1 ]
```

A simple game in Rebol

We're now going to write a short Rebol program and, in this case, we'll start with a simple game. It is a matter of guessing a number selected by the computer. The game guides us by indicating if our guess is higher or lower than the secret figure.



```
NEW GAME
Your figure ?50
Too small !
Your figure ?75
Too small !
Your figure ?95
Won in   3  tries
NEW GAME
Your figure ?
```

Figure 2-2. *Running the game.*

The logic consists of two embedded loops which respectively execute the code indefinitely and ask the player to guess until he finds the correct number. Once the guess is correct, the boolean `end` takes the value `true`, which stops the execution of the loop defined by the word `until`. The secret number is obtained from the word `random` followed by a maximum value. In our case, the figure lies between 0 and 100. We also have a counter, initialised to 0 at the start of an attempt, which indicates how many guesses the player took to win. It is incremented by 1 for each guess.

```
REBOL []
forever [
      value: random 100
      counter: 0
      end: false
      print "NEW GAME"
      until [
            nbr: to-integer ask "Your figure ?"
            counter: counter + 1
            either (nbr <> value) [
                  either (nbr < value) [
                        print "Too small !"
                  ] [ print "Too big !" ]
            ] [
                  print [ "Won in  " counter " tries" ]
                  end: true
            ]
            end
      ]
]
;; Note press the escape key to finish playing
```

Functions and objects

Programming in Rebol consists of defining words in order to extend the dictionary. These words can represent three different components of the language. You already know variables whose role is to store data. The two other types allow the development of more consequential applications; they are functions and objects.

These types of words allow you to structure your code in order to facilitate legibility and its re-use.

A function is a grouping of instructions that carry out one or more definite tasks and are generally executed several times when the program is run. An object contains both methods and properties, i.e. functions and variables respectively. Functions are the building blocks of Rebol; you effectively extend the language by adding functions.

One characteristic of an object is to behave like a black box that performs a specific role well and that can be used repeatedly to save work. An object is a reusable component; its user doesn't need to know its inner workings, only the various entry points, which are known as properties, public methods, and exit points.

Using functions and objects

Whilst a function is usually very dependant upon the environment in which it is executed, an object, on the other hand, is an integrated module that can be used in any project, a way to capitalise on existing code that has been tested and is reliable.

Object-oriented programming makes it possible to consider large-sized developments and teamwork: each team member being responsible for a well-defined part of the program.

As in other languages, functions can be grouped in libraries. Objects can include other objects as well as properties and methods, thus facilitating the intuitive construction of hierarchies of objects. It is also important to immediately learn of a crucial concept in Rebol: contexts. We have already seen that Rebol makes it possible to define the scope of a word. By default, all the words in the dictionary belong to the global context; they are known to and useable by all other words in the same context. On the other hand it is possible with the word `use` to define evaluation contexts different from the global context. In this case, a word cannot use a word belonging to another context. How are the words defined in functions or objects registered? Be very careful with your first Rebol programs because unlike many other languages, all variables defined in a function, unless they are specifically stated to belong to a local context, are automatically placed in the global context of the script.

The opposite is true of objects that are designed to provide independence and modularity and automatically have their own context. A method or a property exists only within the object itself. Any references to these elements must specify the object that contains them.

Defining functions

Rebol makes little, if any, distinction between code and data. A function is simply a kind of data, not surprisingly of the function! datatype which takes as its parameters a list of words corresponding to the values sent to the function when it is called and the code to be evaluated. We can define a function called `square` which, as you will have guessed, returns the square of a number like this:

```
square: make function! [ number ] [
      number * number
]
```

The first block contains the list of parameters, the second the sequence of code to be evaluated by the function. Our function is included in the global dictionary and can now be used by all the words in the language. Which we call by simply typing: `square 4` in the console.

The value returned from a function is always the result of the last evaluation performed by the function. If you want to force a return from a function, i.e. to return a value and stop executing the function, you can use the word `return`.

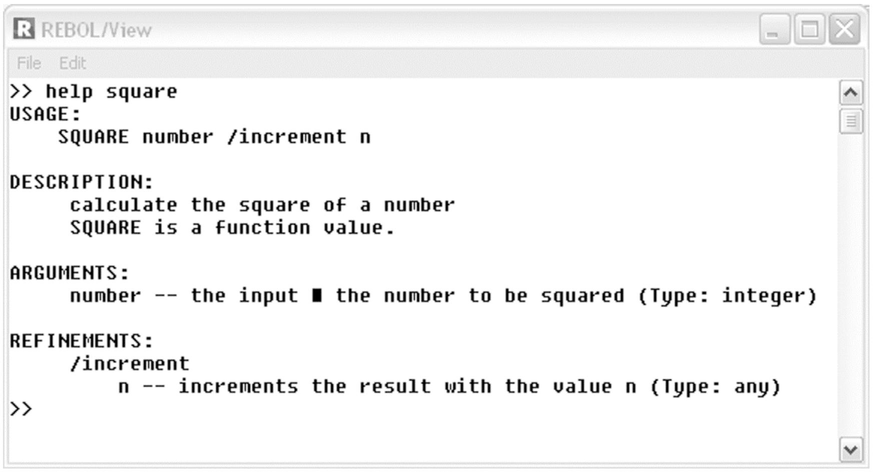
```
square: make function! [ number ] [
      return number * number
]
```

If you want, you can define variables belonging only to the context of the function by using the word `use`. Any word defined in this way will only exist within the function:

```
square: make function! [ number ] [
      use [ result ] [
            result: number * number
      ]
      return result
]
```

With the word `function`, Rebol makes our work easier. This word lets you define in one step, input parameters, local variables and refinements with which you can introduce optional behaviour into a function. It also allows you to specify the datatypes of the parameters and provide information about the function that can be looked up from the console by using the word `help`.

```
square: function [
      "calculate the square of a number"
      number [ integer! ] "the input – the number to be squared"
      /increment n "increments the result with the value n"
] [
      result [ integer! ]
] [
      result: number * number
      if increment [ result: result + n ]
      return result
]
```



```
>> help square
USAGE:
    SQUARE number /increment n

DESCRIPTION:
     calculate the square of a number
     SQUARE is a function value.

ARGUMENTS:
     number -- the input ■ the number to be squared (Type: integer)

REFINEMENTS:
     /increment
         n -- increments the result with the value n (Type: any)
>>
```

Figure 2-3. *Help lets you explore words.*

Creating objects

Defining and using objects in Rebol is extremely intuitive. An object has the `object!` datatype. Its initialisation block simply contains the methods and properties of the object. The definition of a property uses the same rules that you would use for a variable. Writing a method is the same as writing a function. An object may also contain other objects.

The principle semantic difference between objects in Rebol and many other languages is that you use the character "/" instead of "." to specify the access path to a property or method. However, it is similar to such other languages in that an object can refer to itself through the use of the word `self`.

We're going to define an object `computer` whose property `user` defines the first name of the user. The sub-object `hardware` allows the definition of the computer's characteristics and the methods `on` and `off` are intended to switch the machine on or off.

```
computer: make object! [
      user: make string! "Olivier"
      hardware: make object! [
            processor: make string! "Alpha"
            memory: make integer! 32
      ]
      on: make function! [] [ print "I'm switched on" ]
      off: make function! [] [ print "I'm switched off" ]
]
```

In Rebol, the object is immediately usable. You can immediately access the object without going through a process of instantiation, you work directly with the object model (programming by prototype). You can replace or modify the value of the property `computer/user`, the methods are addressed with the syntax `computer/on` or `computer/off`. The object `hardware` needs the complete description of its access path, for example `computer/hardware/processor`. You can also create an instance of the object in the following way: `my-machine: make computer []`.

Now let's create a machine used by Nicolas and fitted with a network card:

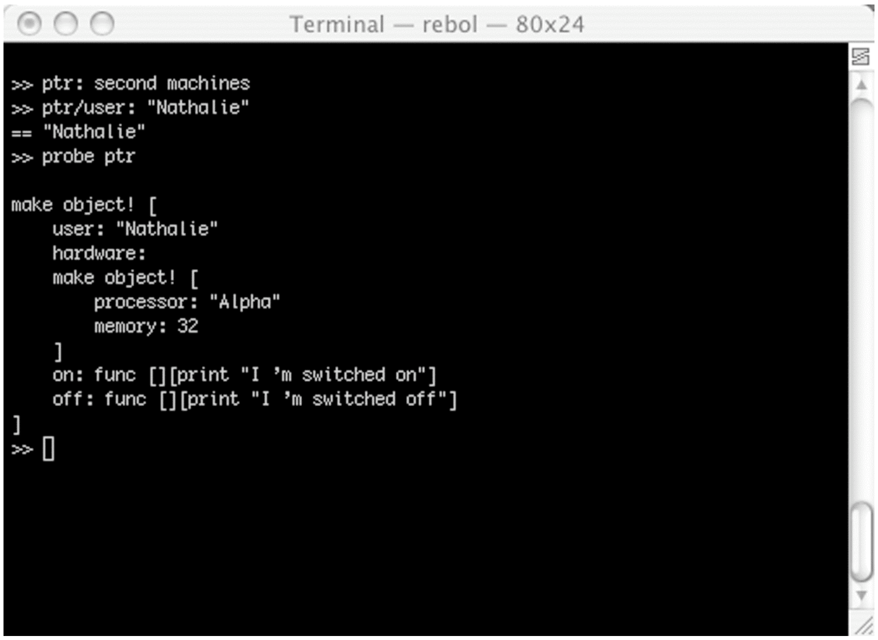
```
server: make computer [
      user: "Nicholas"
      hardware: make object! [
            processor: "X86"
            memory: 128
            network: make logic! true
      ]
]
```

There we inherited from the object `computer` and added a new property to those we had inherited. As you can see from the following code, you can also build lists of objects dynamically:

```
userlist: [ "Olivier" "Nicholas" "Damien" ]
machines: []
foreach the-user userlist [
      append machines make computer [ user: the-user ]
]
```

Afterwards if you want to look-up or modify one of the objects in the list, you'll need to define an object to use as a pointer to the object in the list:

```
ptr: second machines
ptr/user: "Natalie"
```



```
>> ptr: second machines
>> ptr/user: "Nathalie"
== "Nathalie"
>> probe ptr

make object! [
    user: "Nathalie"
    hardware:
    make object! [
        processor: "Alpha"
        memory: 32
    ]
    on: func [][print "I 'm switched on"]
    off: func [][print "I 'm switched off"]
]
>> []
```

Figure 2-4. *The content of objects is displayed by probe.*

Parsing and dialects

We will now take a look at two complimentary aspects of the language: parsing and dialects. If Perl and JavaScript users already know the first, the second is new and is one of the strong points of Rebol. In combination the power for handling character strings and the ability to define your own dialects, true languages in themselves, make a Rebol unique and profoundly expressive tool.

The art of handling character strings

It's no secret that handling character strings is repetitive, painful and tiring. We are all regularly faced with writing algorithms needed to extract from data held in a flat file or a page HTML, to verify a string conforms to a precise specification, etc.. The majority of traditional languages offer the programmer only limited functionality to add characters or to search for one string within another. Happily for us, some developers had had enough of repeating these intellectual contortions indefinitely.

So they the idea that the handling of strings could be modelled in a set of rules: it is the parsing technique. Larry Wall's Perl, originally intended for the data retrieval and report generation, is one of the most famous languages in the parsing domain. In a few characters, it is possible to define an action that can be applied to all the characters in a string.

Rebol parsing

Rebol is a *messaging language*. Its premier function is to obtain, process and transmit information. On the web, most data is being stored within HTML pages or XML documents which are both text files. It is logical that Rebol has a tool for the analysis and extraction of information held in character strings.
The vision of parsing in Rebol is much more modern than that implemented in the majority of the other languages. It is not a question of keying obscure character combinations but of using a true language conceived for this type of operation: a dialect.

Parsing using a dialect

A dialect is a specific language that is integrated with Rebol. It is a language within a language. It is dedicated to a specific task and does only that. If you've come across the term domain specific language, a Rebol dialect is one.

Soon we'll look at the creation of graphical interfaces with the help of Rebol/View. We use the VID (*Visual Interface Dialect*) to define the graphical components of an application. This is probably the first parse dialect that you'll see though you should know that Rebol includes several other dialects

What is so good about dialects is that we can not only use powerful specific languages in their domain but also define our own. It is even possible to implement a BASIC interpreter in Rebol, allowing the two languages to be combined in a single program. In fact, John Niclasen has done just that and implemented a BBC Basic interpreter in Rebol. You can find it at http://www.fys.ku.dk/~niclasen/rebol/bbcbasic.r

But that's not all. Dialects have more of interest: allowing an application to be divided into four parts:

- an engine using a dialect,
- application configuration,
- visual aspects of the application (graphical components),
- application functionality (management rules, help descriptions, etc).

If in current applications we have the opportunity to parameterise the software (maximum values, default values, etc), thanks to dialects, we can now extract the application logic from its body. Setting parameters now extends to its internal logic with its behaviour. The main difficulty then becomes defining a specialised dialect which will be sufficiently general-purpose.

A little parsing

In Rebol, you use the word `parse` to perform a parse operation on a character string. The two parameters for this word are the character string to be manipulated and a block of processing rules to be applied. If the operation consists only of splitting out the string based on a specified character separator, we supply, in the place of the block, only the separator or a character string containing more than one character separator. Using the value `none` as the second parameter indicates that the string must be split by spaces and other standard separating characters such as the comma.

Suppose we have a character string `txt` that contains "A few words on Rebol ", We can separate the individual words from each other by using the expression `parse txt none`. We receive a block that contains a different character string for each word: ["A" "few" "words" "on" "Rebol"].

For our second example, we have a string `txt` that contains the characters "abcdefg". We can split the string by using the characters "b" and "e" as separators. All we need to do is use `parse txt "be"` to get the desired result: ["a" "cd" "fg"].

It is also possible to verify that a string conforms to a specific model. Does the following string contain only two sequences of the characters "xx"?

```
txt: "xx xx"
parse txt [ 2 "xx" ]
```

There we used a rule checking the presence of two "xx" strings inside another string. The word parse then returns a boolean value, `true` if the string conforms to the parse rule, otherwise `false`.

We can also extract data from a string using rules. For example, if we want to extract the characters present between "xx" and "yy":

```
txt: "some xx words yy on Rebol"
parse txt [
      thru "xx"
      copy extract
      to "yy"
      ( print extract )
]
```

Here we split out the contents of `txt` from "xx" until "yy". The result is stored in the word `extract` that is displayed by `print`. The parse dialect allows the execution of Rebol code inside its own code.

Defining a dialect

Creating a dialect in fact means to define analysis rules. If the syntax is correct, the dialect's instructions are executed. For example we can set up a dialect named `cursor` to manipulate the cursor of the Rebol console. We will define two instructions:

- `cls` clears the screen
- `at` positions the cursor at the position indicated by a value of the datatype `pair!` (e.g. 3x5)

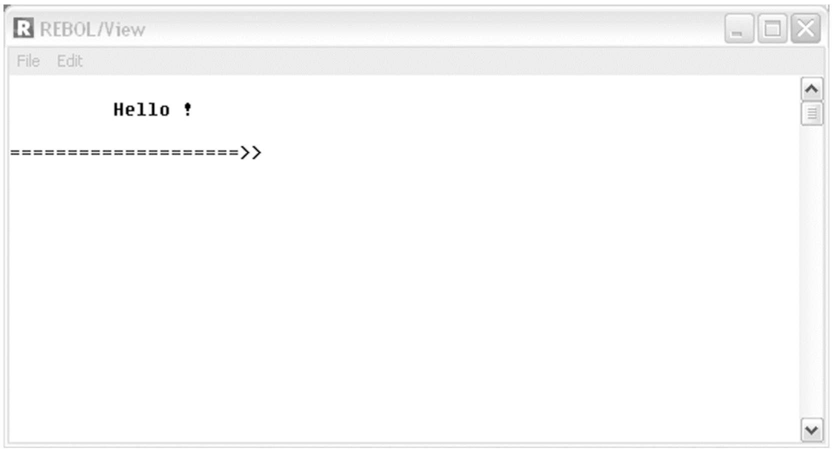
The rules are simple. If Rebol finds the word `cls`, it clears the screen. Otherwise, if the word `at` is found, it must be followed by a value of the type `pair!`. The rules are exclusive. The instruction performed is one of those that were defined (any) but there is only one instruction that relates to each word in the dialect.

For that, we use the "|" character which corresponds to “or”. A function named `cursor` takes a block of code as a parameter to be analysed and parsed. Using the word `compose`, all expressions encloses in parentheses are evaluated and replaced with their value. This functionality allows us to insert variables or Rebol code within code written in this dialect.

```
REBOL [
      Subject: "cursor dialect"
      Author: "Olivier Auverlot"
]

cursor-rules: [
      any [
            'CLS (
                  print "^(1B)[J"
            ) |
            'AT set pos pair! (
                  prin join "^(1B)[" [ pos/y ";" pos/x "H" ]
            )
      ]
]

cursor: function [ code ] [] [ parse (compose code) cursor-rules ]
```

We can now use our dialect in our Rebol scripts. The following example clears the screen, displays a string at the top of the screen and then displays a line of 20 characters underneath it:

```
cursor [
      cls
      at 10x2
]
prin "Hello !"

p: 1x4
for x 1 20 1 [
      p/x: x
      cursor [ at (p) ]
      prin "="
]
```



```
        Hello !

====================>>
```

Figure 2-5. *Cursor dialect test.*

Summary

Rebol is an incredibly simple yet powerful language. It makes it possible to carry out complex operations with the minimum of code. It has many built-in datatypes and facilitates writing very structured code. It also supports an object-oriented approach. Finally, its capabilities in the fields of parsing and dialecting make it profoundly different from other languages.

3
GUI, graphics and sound

In the first chapters of this book, we mainly used the Rebol/Core version in the examples. Being limited to displaying text, that version doesn't allow you to create any really interesting presentation effects. To build graphical interfaces, drawings and generate sounds, there is a version called Rebol/View.

Rebol/View is an extension of Rebol/Core. It does everything that Rebol/Core does but also allows the design of graphic interfaces that are completely platform independent. Your application does not require any changes to work on a different system from the one on which it was developed. In fact, there are even two versions of Rebol/View. The first, called simply Rebol/View, is totally free. You can consider it to be a graphical version of Rebol/Core.

The second, called Rebol/View/Pro, is the commercial version of the language. If you buy it, you then have Rebol/View enhanced with many extensions such as:

- calling functions from dynamic libraries (DLL under Windows, .so files in Linux, etc.),
- powerful data encryption functions such as RSA, DH or DSA.

GCS and VID

To display graphical components on the screen, Rebol/View contains a *Graphical Compositing System*. This very powerful engine not only allows you to display graphics, text and images but also to apply special effects such as colour graduations, mirror effects or modify the colour palette. One can do this working directly with the GCS but then you have to do everything by hand. The GCS represents the lower level functions of Rebol. It is best to reserve their use for very specific applications such as the design of personalised graphic components or the development of complex multimedia products.

Happily for you, the designers of Rebol/View foresaw the need to avoid having to reinvent the wheel with each application. The solution resides in using a dialect called VID (Visual Interface Dialect). This makes it possible to display and handle the principal graphical interface elements with just a few instructions. It is important to remember that VID is just one dialect amongst others. The great idea of Rebol/View is that everyone can develop their own graphic dialect adapted to their own needs (business applications, video games, interactive kiosks, etc.). VID is a generic dialect which allows the very easy development of all types of application.

Basic concepts

The definition of a window's contents is described as a list of settings with the name of `layout`. It is actually a block of instructions for the VID dialogue.

You can also mix Rebol code with the declaration of the different graphic elements. Let's start with a simple example; the following code displays a window with the text "Hello" inside it:

```
view layout [ vh1 "Hello" ]
```

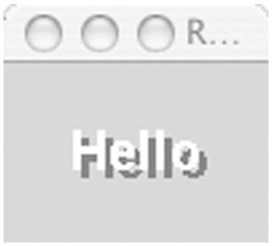


Figure 3-1. *A first window under Mac OS X*

It takes only one line of code to create a window with a text box inside it. Now suppose you want to add a "Quit" button to let the user close the application, the code then becomes:

```
view layout [
      vh1 "Hello"
      button "Quit" [ quit ]
]
```

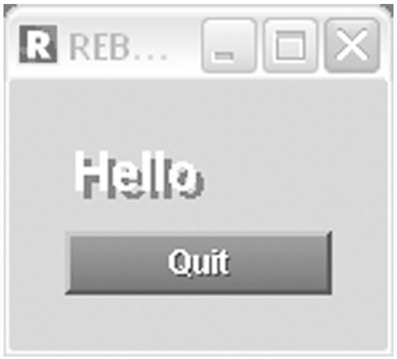


Figure 3.2. *A button added.*

The instructions `vh1` and `button` are called styles in the official Rebol documentation. You can translate this term to the words "component", "graphic element" and possibly even "widget". Rebol speaks about style because the form and behaviour of such components are completely modifiable by the user. The code for handling events is placed in a Rebol code block. This code block can contain several lines and call functions.

Styles

The VID dialect provides a number of predefined graphic elements. There are seventeen just for displaying text. The most practical are `text` for a simple label, `title` and `vh1` for titles. The word `key` lets you capture that a user has pressed a key. The style `button` displays a button.

The styles `toggle`, `rotary` and `choice` are a button with two positions, a rotary button and a drop-down list respectively.

The `check` and `radio` styles define a check box and a radio button. `field` are `area` input fields.

The styles `list` and `text-list` let the user select one or more items from a list. The `slider` style provides sliders. You can also use standard dialog boxes such as a file selector or input boxes or user confirmation request.

In the graphics arena, the style `image` displays the BMP, GIF, JPEG and PNG formats and applies special effects to them. The DRAW dialect draws geometric shapes such as lines, circles, etc. It also provides graphic element transparency that shows the excellent opportunities in the video game field with Rebol/View, especially in networked games.

All these styles have properties so that you can dynamically change their appearance. So that the changes are reflected on the screen, they must be re-displayed with the word `show`.

Attributes

In fact, all the graphic elements of the VID use the same parameters and they can be placed in an unspecified order.

According to the type of data, VID is able to know for which attribute a parameter is intended:

- a character is text to be displayed on the screen,
- graphic co-ordinates (the `pair!` datatype) indicate the dimensions of the object,
- a `tuple!` gives the colour of the element to be displayed,
- a file corresponds to an image,
- a character is a keyboard shortcut,
- a block is a sequence of code to be executed on receipt of an event.

Style layout

There are two ways to place graphic components at a certain position:

- you indicate a fixed position for each component This method is not recommended at all (the many platforms on which Rebol/View don't all have the same graphic resolution),
- you define layout strategies. Then it is no longer a question of where to specify the elements location but how they are to be placed on the screen. For example, `across` indicate that the components should be placed horizontally, one after another. On the other hand, `below` requests the components to be displayed vertically. It is also possible to define panels to group together graphic related components.

A dollar-euro converter

The following example is a dollar-euro converter. The complete code takes only 628 bytes.

```
REBOL [
      title: "Dollar/Euro converter"
]

convert: function [ value /dollar /euro ] [ sum ] [
      sum: to-decimal value
      either dollar [
            sum: sum / 1.30
      ] [ sum: sum * 1.30 ]
      price/text: copy (to-string sum)
      show price
]
```

```
view layout [
      price: field 200x20 ""
      across
      dollar: radio of 'currency true [ convert/dollar price/text ]
      text "Dollar"
      euro: radio of 'currency [ convert/euro price/text ]
      text "Euro"
      return
      button "Quit" 255.0.0 [ quit ]
      button "Calculate" [
            either dollar/data = true [
                  convert/euro price/text
            ] [ convert/dollar price/text ]
      ]
]
```

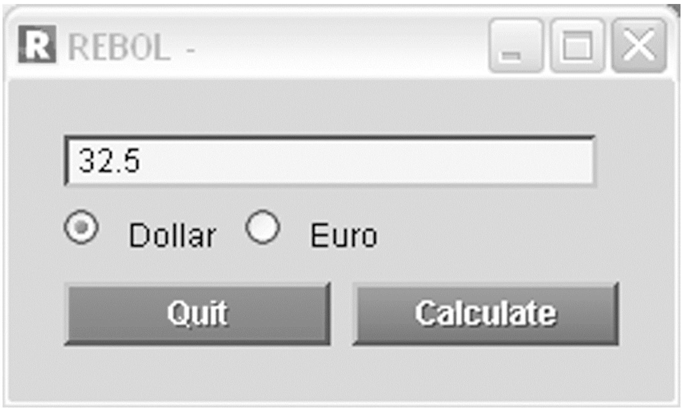


Figure 3-3. *A fully functional converter.*

The small size of applications developed with Rebol/View enables them to be easily downloaded over networks.

Image processing with VID

We will continue our discovery of Rebol/View's Visual Interface Dialect by using its graphical capabilities. VID is not limited to displaying buttons and input fields, it also let you display images, change them and apply complex special effects to them.

Using images

Rebol/View can use four of the most common image formats. First of all there is Microsoft's BMP format. Compatibility with this format allows you to make use of a great number of existing images. In addition to this Windows standard, Rebol/View support the two most widely used formats on the web, JPEG and GIF and, also, the newer PNG format, conceived to replace the GIF.

If an image is in one of these four formats, it only takes a single instruction to load it into memory and make it available. We simply use the word `load` followed by the file name. The access path of the file can be either be local or a URL allowing an image to be fetched across a network. You have to handle any other image types yourself (which is very complicated – so it is best to use the supported types if at all possible).

```
myimage: load %tuxrebol3.gif
```

Now, the image has been converted to Rebol's internal format in the form of a list of binary values which can be looked at with the help of the word `mold` and can be modified like any other series value in Rebol.

Displaying an image

Using the instruction `image` of the VID dialect, we can display the image in a window on the screen. It isn't even necessary to load the image in memory before displaying it. In effect, if we supply a filename as a parameter to the instruction `image`, it will retrieve the image from the file before displaying it. On the other hand if the attribute is a variable containing an image, the instruction will display it directly. In fact, it simply depends on whether we want to store the image in memory. The following script displays an image directly on the screen.

```
view layout [
      image %tuxrebol3.gif
]
```

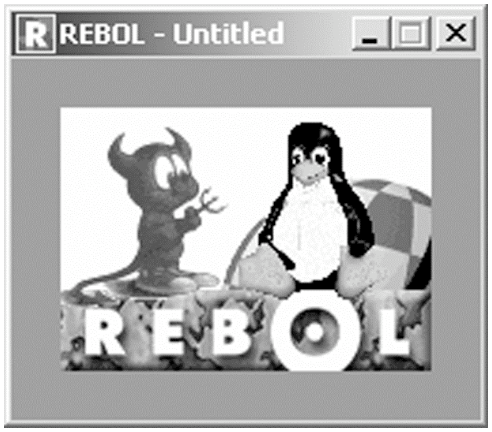


Figure 3-4. *Displaying an image.*

It takes only a few lines of Rebol code to build a powerful visual display of images. Let's see how easy it is to select images in a dialogue box and independently display them in other windows in Rebol/View.

We create a `layout` containing two buttons. The first allows the choice of an image; the second closes the application.

The dialogue to select a file is a standard function in Rebol/View and is called `request-file`. (There is a set of standard dialogues that you can find out about by typing `help request-` in the console.) The `filter` refinement filters the files that are displayed in the dialogue box. Here the user can only select BMP, GIF, JPEG or PNG images. If the user clicks "Cancel", `request-file` returns the value `none`.

As the user has the ability to choose several files by using the CONTROL key, we must display all of the images selected by using a `foreach` loop. For each file chosen, the image is displayed in a window with a black background:

```
REBOL [
      subject: "image display"
      author: "Olivier Auverlot"
]
```

```
view layout [
      button "Load" [
            choice: request-file/filter [
                  "*.png" "*.gif" "*.jpg" "*.bmp"
            ]
            if not none? choice [
                  foreach file choice [
                        view/new layout [
                              backdrop 0.0.0
                              image file
                        ]
                  ]
            ]
      ]
      button "Quit" [ quit ]
]
```

Modifying images

Rebol/View can do much more than simply display images, it also allows you to dynamically change them and apply complex special effects to them.

An image can be resized simply by providing the word `image` with the size of the image as a `pair!` value.

We can apply a colour onto the image by simply supplying a RGB value as a `tuple!`. Suppose we want to dye the image `tuxrebol3.gif` red and fix its size at 50 by 50 pixels, the syntax would be:

```
view layout [
      image 50x50 255.0.0 %tuxrebol3.gif
]
```

Now suppose that we want to reduce the image to 50% of its real size, then we must determine its new size based upon its original size. For this, we use the `size` property of the image.

```
myimage: load %tuxrebol3.gif

new-size: make pair! reduce [
      (make integer! (myimage/size/x / 100) * 50)
      (make integer! (myimage/size/y / 100) * 50)
]

view layout [ image new-size myimage ]
```

Applying special effects

Without question the most visually impressive feature of Rebol/View is its capacity to apply a whole host of special effects, inspired by the most powerful image improvement software such as Photoshop or Gimp, not only to an image but also to all graphic components (buttons, lists, edit fields, …).

So we can apply transparency, modify the colours, resize, crop, zoom, rotate, mirror, change the luminosity, generate colour graduations or generate relief effects. All this functionality is included in standard Rebol/View and is available on all of the supported platforms. Multimedia programmers or video-game makers have a new generation tool which is equally at home networking and processing data as it is with animation and graphic special effects. Rebol/View is well positioned in the markets for networked games, interactive CD-ROMs and electronic terminals. For the first time, such products will be compatible across multiple platforms without modification.

Special effects are activated by the `effect` attribute of a style. In the examples we will use an image but remember that these operations can be carried out to any graphic component. For the following examples, we will torture Tux! The image `tux.gif` uses a blue base (0.0.255) to fix the transparent zones. Let us start by placing it on a base coloured by applying a transparency effect using the `key` attribute.

```
view layout [
      backdrop 96.128.128
      image %tux.gif effect [ key 0.0.255 ]
]
```

Now we will give it a good shaking.

With the help of the `rotate` attribute, we can rotate an image. By using a `slider`, the user can select the angle applied to poor Tux:

```
view layout [
      backdrop 96.128.128
      tux: image %tux.gif effect [ key 0.0.255 rotate 0 ]
      pos: slider 100x20 [
            tux/effect/rotate: pick [ 0 90 180 270 ]
              ((make integer! (pos/data) * 3) + 1)
            show tux
      ]
]
```

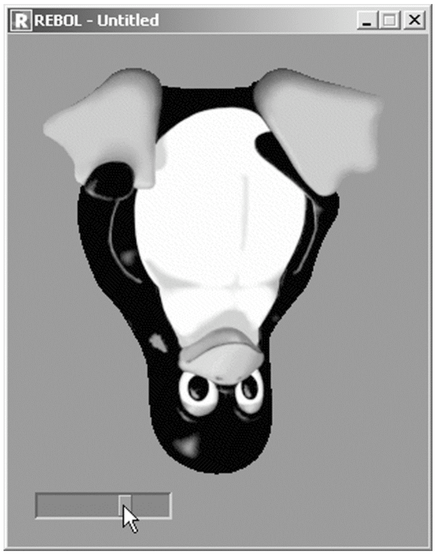


Figure 3-5. *Rotating an image*

It is not really any more difficult to apply a relief effect to the Linux mascot. For this, we are going to define a check-box "Relief" which depending on the current state adds or removes the `emboss` effect applied to the image.

```
view layout [
      backdrop 96.128.128
      tux: image %tux.gif effect [ key 0.0.255 rotate 0 ]
      across
      pos: slider 100x20 [
            tux/effect/rotate: pick [ 0 90 180 270 ] ((make
      integer! (pos/data) * 3) + 1)
            show tux
      ]
      relief: check "Relief" [
            either relief/data = true [
                  append tux/effect [ emboss ]
            ] [ remove (find tux/effect 'emboss) ]
                  show tux
      ]
      text "Relief"
]
```

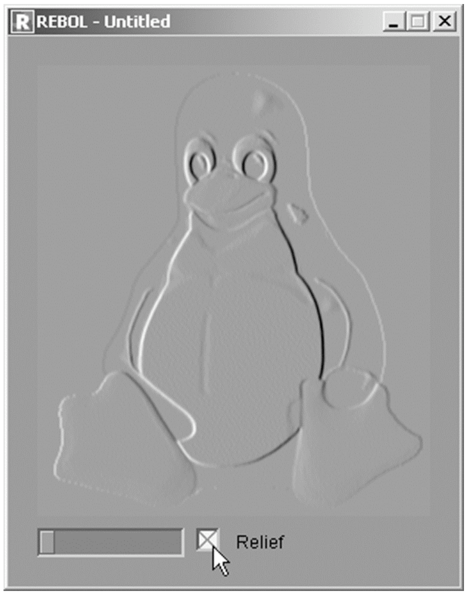


Figure 3-6. *Applying an effect.*

Rebol/View has about thirty different effects which support the creation of animations and complex graphic handling.

Once again, the power and speed of the Rebol/View GCS (Graphical Compositing System) makes it a single, general-purpose tool. Whilst its competitors require writing of hundreds of lines of code and the use of additional libraries, Rebol is extremely concise and self-contained.

The DRAW dialect

Let's continue to explore graphic programming with Rebol/View. We will now look at the functions to draw geometric shapes on the screen. These functions are part of the DRAW dialect which is an integral part of the `effect` attribute of each graphic component. This may appear a little off-putting or over-complicated but it is actually very powerful.

Let's draw a line

The DRAW dialect is fact part of VID as a sub-dialect. You can only use it by including it in the `effect` attribute of a style.

Contrary to VID, the order of the parameters is not specified: any substitution or inversion of an attribute generates an error or causes the DRAW code not to run.

You place the DRAW instructions in any style, even on top of an image or button. In general, the most practical style for drawing is `box` because it makes an excellent drawing board and can also be equipped with a *timer*. Thanks to that, it is really easy to produce animations.

The colour drawn is defined by the `pen` instruction followed by the name of the colour or its RGB code (3 octets representing red, green and blue).

To draw a line or place a dot, you use the `line` instruction followed by two co-ordinates, as `pair!` types, for the start and end points. The co-ordinate axes are Cartesian with the origin classically placed at the top left of the component.

The instruction `line-pattern`, with appropriate `pen` settings, lets you change the appearance of the line by using dashed and dotted lines. Before you specify the pattern of the line, you must first set the `pen` to the two colours in which you want the line to be drawn. Usually, we set the line to be a single colour so we specify the first of the two colours to be `none`. This sets the first colour to be totally transparent. (Be careful because this only works if `none` is specified before the other colour). You then supply `line-pattern` with an even number of integer parameters.

When you next draw a `line` the dialect uses the first colour for the number of pixels specified by the first parameter, the second colour for the number of pixels specified by the second parameter, then the first colour for the number of pixels specified by the third parameter, and so on. (At the time of writing, the Draw dialect only applies up to a maximum of four parameters.)

```
view layout [
      box 100x100 effect [
            draw [
                  pen white
                  line 0x0 100x100
                  pen none white
                  line-pattern 4 2 5 10
                  line 0x100 100x0
                  pen white
            ]
      ]
]
```

Other draw functions

You have a complete set of drawing instructions at your disposal. You can draw a rectangle with the word `box` followed by two co-ordinates (of datatype `pair!)`, the first is the top left-hand point of the rectangle, the second the bottom right-hand corner.

The `circle` command draws a circle defined by its centre and radius.

To fill a shape with colour, you simply use the `fill-pen` instruction followed by a colour before drawing the shape. You can actually specify `fill-pen` to use very complex colour graduations by supplying optional parameters. (If you want an empty shape once you have set the `fill-pen`, you must specifically set it to `none). polygon` is a powerful instruction that plots straight-lines between the given co-ordinates. The geometric shape is closed as `polygon` automatically joins the last point to the first. You can fill the shape by simply set the `fill-pen` before drawing it.

```
view layout [
      box 300x200 effect [
            draw [
                  pen white
                  box 10x10 100x100
                  circle 150x50 30
                  polygon 200x10 290x60 270x90 210x110
                  fill-pen red
                  circle 150x140 40
                  box 20x140 100x190
            ]
      ]
]
```

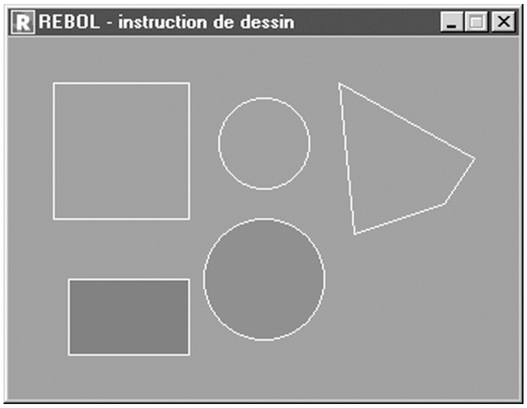


Figure 3-7. *Using the DRAW dialect.*

Adding text

The instruction `text` displays a character string at a specified co-ordinate. The colour of the text is fixed by the instruction `pen`. It is also very simple to create specially effects such as shading the text:

```
view layout [
      box 200x100 effect [
            draw [
                  pen white
                  line 0x0 200x100
                  line 0x100 200x0
                  pen black
                  text 2x42 "Text added with DRAW"
                  pen red
                  text 0x40 "Text added with DRAW"
            ]
      ]
]
```

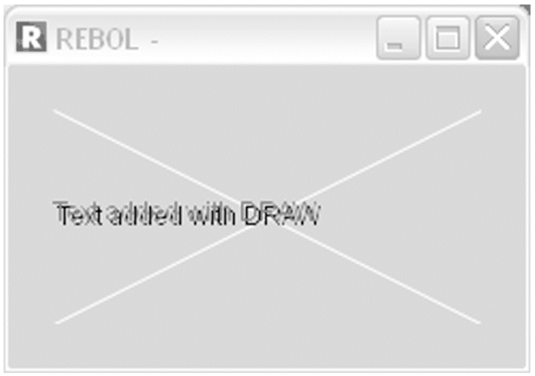


Figure 3-8. *Displaying text with DRAW.*

Manipulating images

In a drawing zone, you can also display BMP, GIF, JPEG and PNG images with the help of the `image` instruction. You only need to provide a word containing the image itself and the Draw dialect will do the rest. If you'd prefer a little more control you can specify the top left-hand co-ordinate, the top left and bottom right co-ordinates or the co-ordinates of all four corners to specify where the image will be displayed. You can also supply the RGB code for a colour to be rendered transparently.

```
img-tux: load %tux.bmp
monster: load %monstre.bmp

view layout [
      box 170x180 effect [
            draw [
                  image img-tux 0x0
                  image monster 30x96 0.0.255
            ]
      ]
]
```

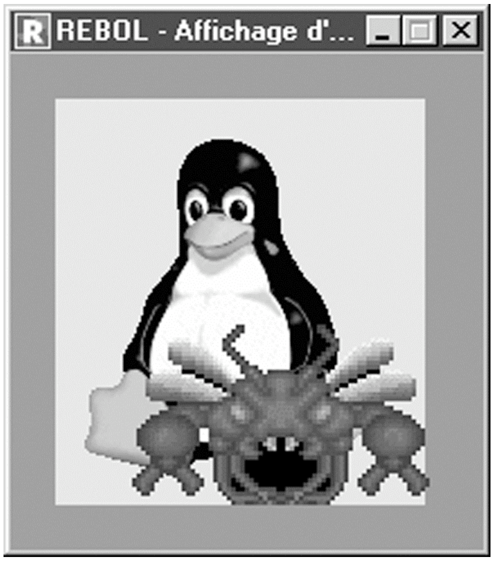


Figure 3-9. *Adding an image with transparency.*

This ability to manipulate images enables you to very easily create sprites, superimposed images that appear on the screen in front of a background. This functionality is very helpful when it comes to developing video games.

Generating image files.

When you have finished your drawing, you can generate an image from it. Such an image can remain in memory and be used in the same manner as any other image (apply effects, display, assigned to a style, etc.). It can also be saved to disk in one of two formats, PNG or BMP. You only have to use the word `save` with the correct *refinement* (`/bmp` or `/png`) and to convert your drawing into an `image!` type. You end up with a file that can be read by other software.

```
view layout [
      mybox: box 100x100 effect [
            draw [
                  pen white
                  line 0x0 100x100
                  line 0x100 100x0
            ]
      ]
      button "save" [ save/png %test.png (make image! mybox) ]
]
```

Using DRAW dynamically

Until now the examples we have seen have been very static. You are probably asking yourself a number of questions: how to plot a series of figures on a graph or how to make a sprite move within a frame? Don't panic! VID and DRAW are extremely flexible.

Generating DRAW instructions

In fact, DRAW instructions are simply a block of data. If you want to dynamically generate instructions for a drawing, all you need to do is to modify the content of the style's `effect/draw` block.

To better understand this, we are going to draw a histogram from a series of numbers. They are contained in a series and represent the height (in pixels) of each rectangle:

```
numbers: [ 100 130 80 110 50 90 ]
```

When the user clicks on the "Trace" button, the script initialises the `graph/effect/draw` block with the colour to be drawn and draws the two axes (x and y). Then a `foreach` loop reads each value in turn and calculates the co-ordinates of the rectangle to represent the data. On each pass of the loop, the `box` is added to the `graph/effect/draw` block. Once this operation has finished, all is left to do is to refresh the style called `graph`. The histogram will then appear on the screen.

```
view layout [
      graph: box 200x200 effect [ draw [] ]
      button "Trace" [
            pos-dep: 10x190
            graph/effect/draw: copy [
                  pen white
                  line 5x0 5x200
                  line 0x190 200x190
            ]
            foreach value numbers [
                  pos-fin: pos-dep + make pair! reduce [
                        20 (value * -1)
                  ]
                  append graph/effect/draw reduce [
                        'box pos-dep pos-fin
                  ]
                  pos-dep: pos-dep + 20x0
            ]
            show graph
      ]
]
```

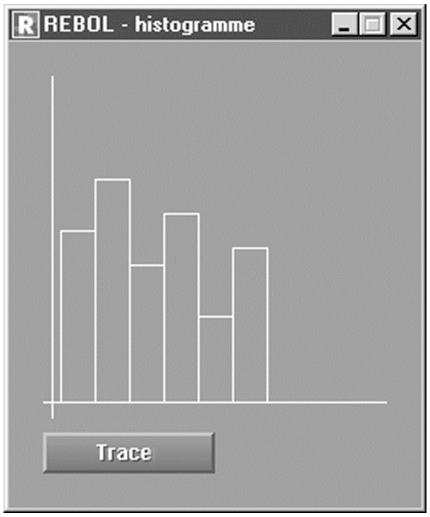


Figure 3-10. *A generated histogram*

.

A little animation

With the help of the DRAW dialect, it is possible to create high-quality animations. One of the reasons for this is that Rebol uses a virtual screen to build screen displays. The word `show` copies this virtual screen to the physical one. So there is no flickering. Moreover, VID offers the programmer *timers* to set-up timeslots. Thanks to them you can, for example, animate sprites every 5 tenths of a second. This makes it possible to achieve animation speeds equivalent to the power of the computer running the script.

In VID, most styles have an integrated, independent *timer*. All that is needed is to use the `rate` attribute to specify a time delay or the number of times per second the style is to be activated. The following example draws a random line ten times a second:

```
view layout [
      mybox: box 200x200 rate 10 effect [ draw [] ] feel [
            engage: func [ f a e ] [
                  if a = 'time [
                        append mybox/effect/draw reduce [
                              'pen (random 255.255.255)
                              'line (random 200x200) (random
                        200x200)
                        ]
                        show mybox
                  ]
            ]
      ]
]
```

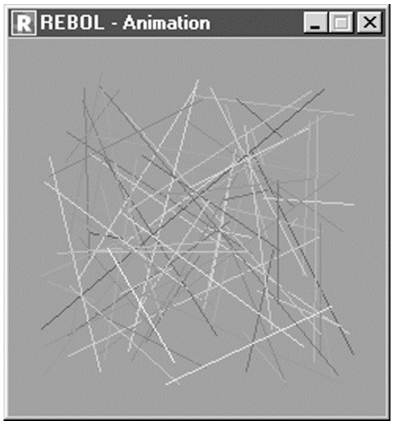


Figure 3-11. *Drawing Random Lines.*

You will have noticed, an event handler must be set up with the `feel` attribute to detect and process a `time` event.

Handling events with VID

The VID dialect supports the easy, quick development of graphic user interfaces that are fully independent from the executing platform. As you have seen, you can easily position and display graphic components such as buttons, text and images on the screen. The next stage is to interact with these components to respond to users' actions. We will now take a full look at event-driven programming with Rebol.

Event-driven programming

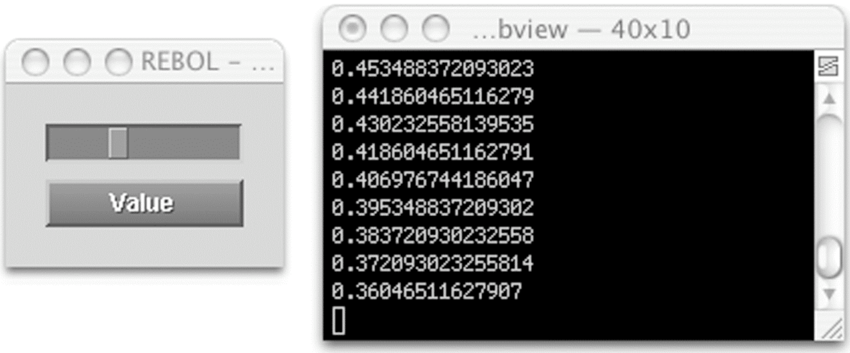
A script using a graphic user interface works by having a loop which waits for events produced by user actions (pressing a key, selecting a graphic component, moving the mouse, etc.) as well as those generated by the system (clock, receipt of data packets from the network, etc.). The heart of a VID application consists of marshalling the correct component's code to execute depending on the event received. The dialect also allows you to completely redefine the behaviour of a component so that you can create new styles specifically adapted to your project.

Default behaviours

Each of VID's many components has a default behaviour. It is a block of code, enclosed between brackets, which is executed for a specific action. A button, for example, evaluates its code when a user clicks on it. A radio button or a check-box also reacts to the user clicking on it. On the other hand, an input box activates its code each time the user presses the Enter key. A slider modifies its value each time it is moved.

In the following example, a window displays a slider and a button. On each change in the position of the cursor or click on the button, the value of the slider is displayed (ranging from 0 to 1) in the console:

```
view layout [
      myslider: slider 100x20 [
            print myslider/data
      ]
      button "Value" [
            print myslider/data
      ]
]
```



```
0.453488372093023
0.441860465116279
0.430232558139535
0.418604651162791
0.406976744186047
0.395348837209302
0.383720930232558
0.372093023255814
0.36046511627907
```

Figure 3-12. *Styles reacting to user actions.*

You probably understand that this method is very simple but it isn't the most powerful. All the same it makes it possible to quickly cater for the basic interaction between a user and an application. To go further, it is necessary to dig a little deeper into the mechanics of event management in Rebol/View.

Tracking events

Each VID object has an attribute called `feel` for handling specific events. The event handler applies to the specific object itself and not to any other object based on the same style. So two buttons or two images can have completely different behaviour. If you decide to modify the behaviour of a style, you must create a new style derived from the original model (often referred to as the prototype).
In the `feel` attribute you can place four functions, each one having a precise role. The `detect` function intercepts all incoming events and distributes them to the other three functions. `engage` lets you detect a mouse-click or a `time!` event. `over` detects when the mouse pointer passes over the component and also when it exits from the component. Finally, `redraw` allows a component to be redrawn.

This last operation is automatic in VID but it is possible to de-activate it in order to improve application performance or so that the programmer can handle particular cases individually.

Each of these functions receives its parameters, which describe the event, directly from Rebol/View's *Graphical Compositing System* (GCS).

The information about the event is passed as an object in the `face` variable. The `action` variable provides the type of event the GCS intercepted. The position of the mouse pointer can be found in `offset`. The description of the event is contained in the `event` object. The boolean variable `over?` indicates if the mouse is inside or outside the component.

The code which follows is a complete skeleton for handling VID `button` events. From this model, you can entirely redefine how it reacts to user actions:

```
view layout [
      mybutton: button "Hello" feel [
            engage: func [ face action event ] [
            ]
            over: func [ face over? offset ] [
            ]
            detect: func [ face event ] [
            ]
            redraw: func [ face action offset ] [
            ]
      ]
]
```

At the moment, your "Hello" button seems inoperative. Actually, you have redefined all of its reactions to user actions to do nothing. Suppose that you want to display some text each time the mouse pointer is over the button or when it is not, all you need to do is to modify the `over` function as follows:

```
over: func [ face over? offset ] [
      either over? [
            print "over"
      ] [ print "outside" ]
]
```

The event object

This object thoroughly describes the event intercepted by the GCS with seven attributes.

You can tell what type of event has been trapped from the standard `type` property. The mouse buttons provide the `down`, `alt-down`, and `up` events. Moving the mouse produces a `move` event. Pressing a key on the keyboard generates a `key` event. User changes to the display windows create `resize`, `close`, `active`, `inactive` and `offset` events. Finally, a *timer* with a regular interval starts a `time` event.

Once you have determined the type of event, you can use other properties of the `event` object, which contain additional information about the event.

The `offset` property returns the position of the mouse. The `key` property contains the value of the key which was pressed by the user. The state of the control and `shift` keys may be found from the two properties bearing the same name as the keys; they contain boolean values. The `time` property is a timestamp of when the event occurred and finally the `face` property is an object containing the component activated in the event.

As you can see, it is very easy to change the behaviour of a VID style. In a few lines of code you can, for example, change the colour of a button when the user moves the mouse over it or when the user clicks the mouse's left button. To do this, you must modify the `engage` and `over` methods of the button and to handle the `down` event (the left mouse button pressed by the user) and the parameter `over?`.

```
view layout [
      mybutton: button "Hello" feel [
            engage: func [ face action event ] [
                  if event/type = 'down [
                        face/color: 0.255.0
                        show face
                  ]
            ]
            over: func [ face over? offset ] [
                  either over? [
                        face/color: 255.0.0
                  ] [ face/color: 0.0.255 ]
                  show face
            ]
            redraw: func [ face action offset ] [
            ]
      ]
]
```

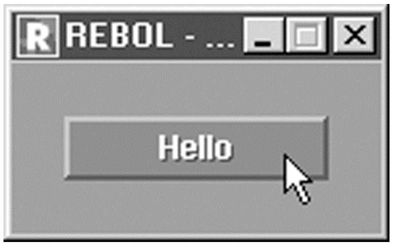
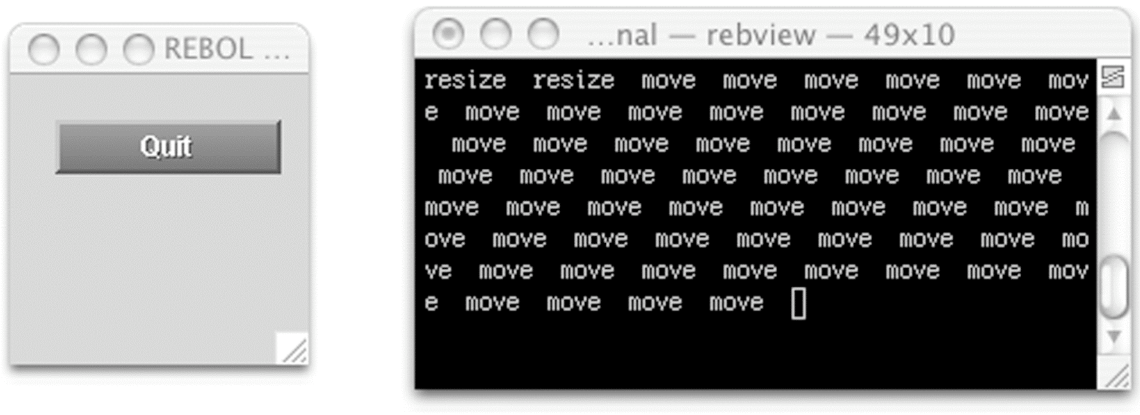


Figure 3-13. *The button colour is changed as the pointer passes over it.*

Controlling Windows

For the windows in your application, the most effective method is to set up a single handler for all of the events that they produce (opening, closing, moving, etc.). With the help of the `insert-event-func` word, you insert a function into the main GCS event monitoring loop. The aim of this is to react to each of the events produced by your application windows. One of two parameters the inserted function will receive is an `event` object which lets you determine which window is affected (`event/face`) and the type of event (`event/type`). The function must end with the word `event` to allow the script to continue executing by starting the processing of any events waiting in the queue. The word, `remove-event-func,` lets you remove an event handler once it is not longer needed. The example below displays all of the events of the window named `wind`:

```
display-events: function [ face event ] [] [
      if event/face = wind [ prin [ event/type " " ] ]
      event
]
insert-event-func :display-events

wind: layout [
      button "Quit" [
            remove-event-func :display-events
            quit
      ]
]
view/options wind 'resize
```



```
resize  resize  move  move  move  move  move  mov
e  move  move  move  move  move  move  move  move
  move  move  move  move  move  move  move  move
 move  move  move  move  move  move  move  move
move  move  move  move  move  move  move  move  m
ove  move  move  move  move  move  move  move  mo
ve  move  move  move  move  move  move  move  mov
e  move  move  move  move
```

Figure 3-14. *Some of the events intercepted.*

Once again, Rebol remains faithful to one of its core concepts: to give total freedom to the programmer. Rebol gives you very fine control over events in your application.

Managing styles

The VID gives a perfect example of just how effective Rebol can be in use. A few lines of code are all that is needed to develop a complex graphic interface. The programmer has the freedom to personalise the numerous standard styles and to add new ones. These elements can be gathered together in true style sheets. VID allows the programmer to modify all available styles: you have the means to adapt how they appear so that you can create a unique look for your application.

If VID's presentation is not convenient for you, nothing prevents your from adding a touch of QNX, Aqua or even Windows to your script. Nothing here is cast in stone; it can all be changed at ease.

Defining a style

A style is defined within the `layout` which will use it. The keyword `style` is followed by the name and attributes of the style you wish to create. Your style inherits the properties and methods from a specified model (its prototype) but is different from it where you have specified changes.

To help understand, let's start with something simple: our objective is to define a style called `red-button`. It is similar to a standard button except it is red with white text. You will "derive" the `button` style and change its colours:

```
view layout [
      style red-button button red
      red-button "button1" [ print "button1" ]
      button "button2" [ print "button2" ]
      red-button "button3" [ print "button3" ]
]
```

Once the style has been defined in the `layout`, you can use it in just the same way that you would use a standard VID style. The only constraint is that you must define the new style at the beginning of each `layout` in which it is used. In order to solve this problem, VID supports the creation of style sheets to enable the separation of application logic and display settings.

Applying a style sheet

The principal of style sheets in Rebol is very close to that of CSS with HTML. At the start of your applications or in a separate file, you describe the appearance and behaviour of your personalised styles. You then have the ability to connect one or more style sheets to different `layouts` in your program. A style sheet is defined using the word `stylize` followed by a block containing the description of its different styles.

This block is assigned to a word not only so that it can be identified but also so it can be attached to a layout with the VID keyword `styles`.

Let's take the previous example and add to it another style called `red-text` which will be a simple label with red text. The style sheet bears the name `my-styles`:

```
my-styles: stylize [
      red-button: button red
      red-text: text red
]
```

All you need to do now to use your new styles is to include your style sheet in the layout in which you want to use them.

```
view layout [
      styles my-styles
      red-text "I am written in red text"
      red-button "Button1" [ print "Button1" ]
      button "Button2" [ print "Button2" ]
]
```

This method allows you to use many different style sheets in a single application and, very helpfully, make the look of your software very flexible. The ideal is to store your style sheets in files and then include them with the help of the word `do` when your script is launched. Just modifying these style sheets makes it possible to dramatically change the appearance of software. Organising style sheets this way gives a company the chance to define presentation standards for all its applications or to adapt the look and feel of an application to a customer's requirements.

Modifying style aspects

For a single specific need or in the case of creating a complex new style, you can also make in-depth changes to a standard VID style. So it is possible to change the attributes which define the contour of the style, the character font used to display text or even the characteristics of a paragraph.

To change the border of a style, you must modify the attributes of the `edge` block that exists for each VID component. The `size` attribute is a `pair!` and fixes the width and height of the border of the component. With `color`, you specify the border colour which is drawn as specified in the `effect` attribute. This last attribute takes one of the values (`'bevel`, `'ibevel`, `'bezel`, `'ibezel`, `'nubs`) which determines how the edge will be drawn.

With the block named `font`, you can select the character font (using the keyword `name`) from one of three fonts (`font-serif`, `font-sans-serif`, `font-fixed`). The `style` parameter allows you to choose the style of text (`'bold`, `'italic` or `'underline`). `size` and `color` set the character size and colour.

You can determine how the text will be aligned by using the `align` parameter followed by one of the values `'left`, `'right` or `'center`.

Paragraph formatting is controlled by the `para` facet which takes a block of parameters. You set the precise position at which the text is displayed with the `origin` parameter. With `scroll`, you can allow horizontal or vertical scrolling of the contents of a VID style. The `tabs` parameter can contain one or more values to create tab stops in the text. Finally, `wrap?` is a boolean value to set word wrapping on or off.

Let us look at an example to help understand better: the objective is to create a new style called `big-button`. This one displays its imposingly sized font with a relief effect at the edge. To do this, it is best to start with a simple `box` style. This is probably the most neutral VID style and so can be used for the creation of the majority of new styles. With it you have great freedom in handling events. The definition of this box indicates the font to be used, the text alignment and the appearance of the border.

```
my-styles: stylize [
      big-button: box font [
            name: font-serif
            size: 40
            align: 'center
            valign: 'middle
      ] edge [
            size: 10x10
            color: 192.192.192
            effect: 'bevel
      ]
]
```

Now all you have to do is to include the style-sheet in your layout so that you can use it.

```
view layout [
      styles my-styles
      backcolor 0.0.255
      big-button "Rebol !" 200x100 0.0.0
]
```

Defining a style’s behaviour

In spite of its changed appearance, the style you have defined adopts the behaviour of its prototype. For the moment, your button reacts to any user action in the same way as a standard `box` would.

For it to act like a real button, you have to define event handlers for this new style.

Your objective is that the button is inverted when the left mouse button is clicked and, at the same time, some Rebol code is executed. To achieve that, you must redefine the `engage` method whose role is to signal the click and release of the mouse button.

You change the height of the border by changing the `effect` property of the `edge` facet of the currently active object (`face`). The code passed as a parameter is executed by the Rebol `do-face` function.

```
my-styles: stylize [
      big-button: box font [
            name: font-serif
            size: 40
            align: ‘center
            valign: ‘middle
      ] edge [
            size: 10x10
            color: 192.192.192
            effect: ‘bevel
      ] feel [
            engage: func [ face action event ] [
                  either event/type = ‘down [
                        face/edge/effect: ‘ibevel
                        do-face face none
                  ] [
                        face/edge/effect: ‘bevel
                  ]
                  show face
            ]
      ]
]

view layout [
      styles my-styles
      backcolor 0.0.255
      big-button “Rebol !” 200x100 0.0.0 [
            print "Super big button !"
      ]
]
```

Your style now reacts to user actions and executes the code when it is pressed.

Rebol and sound

After studying the graphical capabilities of Rebol/View, you are now going to discover what it offers in the sound arena. Rebol is able to read and manipulate sounds samples allowing the development of multimedia applications such as games, interactive CD-ROMs and information kiosks. We will learn about this subject by developing a sound player, which will enable us to apply our knowledge of VID.

Opening and closing a sound port

Rebol/View and Rebol/Command include a port called `sound` specifically for playing sounds. This post must be opened with the word `open` and closed with the word `close`. Be wary of this type of port as it can take a little time before it's activated by the system. Don't hesitate to use the word wait to set a short pause (two or three tenths of a second should be long enough) before playing the first sound if you need to do that straight after opening the port. Checking for an error as you `open` the port lets you find out if the version of Rebol you are using has sound functionality. In this way you can modify a boolean value to indicate whether your script is wired for sound. For our example, if we cannot use the sound port, we will simply end the script with an error message:

```
if error? try [
      sound-port: open sound://
      close sound-port
] [
      alert "You don't have access to sound"
      quit
]
```

Loading and handling sound samples

Rebol can handle non-compacted (*WAVeform*) sound samples. A WAV file is loaded into memory by the `load` word which generates an object with five properties which can be both read and written:

- `data` contains the data which will be played by your machine's sound card,
- `volume` defines the output sound volume (between 0 and 1),
- `channels` specifies the number of sound channels (1 or 2),
- `bits` specifies the number of bits used for sampling. If the value is 8 bits, each value of the sample constitutes amplitude between 0 and 255. For 16 bits, the range of values extends from 0 to 65535.
- `rate` indicates the sampling frequency in hertz.

The quality of a sound sample is mainly dependent on the number of bits used and the frequency of them. Sound sampled at 44100 Hz using 16 bits, the norm for an audio CD, will be higher quality than one at 8000 Hz of 8 bits (close to the quality of a telephone line).

We can now start to create the principal screen of our WAV sound player. The `layout` contains a "File" button for loading a sound sample into a word called `echantillon` (French for sample):

```
button "Files" [
      files: request-file/filter "*.wav"
      if all [
            (not none? files)
            (not empty? files)
      ] [echantillon: load first files ]
]
```

With the "Edit" button we can display a dialog box to allow the sound properties to be changed. The user can then change the volume, select a different number of channels, choose a different sampling quality and fix the reading rate while using the sampling rate. The window only appears if a sound is loaded in memory. The fields in the edit box are updated before they are displayed.

```
button "Edit" [
	if not none? echantillon [
		volume/data: echantillon/volume
		either echantillon/channels = 1 [
			can1/data: true
			can2/data: false
		] [ can1/data: false can2/data: true ]
		either echantillon/bits = 8 [
			bits8/data: true
			bits16/data: false
		] [
			bits8/data: false
			bits16/data: true
		]
		freq/text: echantillon/rate
		view/new/title editor "Editor"
	]
]
```

The `layout` named `editor` contains a slider to adjust the volume and an input box to set the frequency. The number of bits and which channel are selected by radio buttons:

```
editor: layout [
	across
	text "Volume :"
	volume: slider 100x20
	return
	text "Number of channels:" return
	can1: radio of 'can text "1"
	can2: radio of 'can text "2" return
	text "Precision:" return
	bits8: radio of 'precis text "8 bits"
	bits16: radio of 'precis text "16 bits" return
	text "Frequency:" freq: field 50 return
	button "Cancel" [ unview edition ]
	button "Update" [
		echantillon/volume: volume/data
		either can1/data = true [
			echantillon/channels: 1
		] [echantillon/channels: 2 ]
		either bits8/data = true [
			echantillon/bits: 8
		] [echantillon/bits: 16 ]
		echantillon/rate: to-integer freq/text
		unview editor
	]
]
```

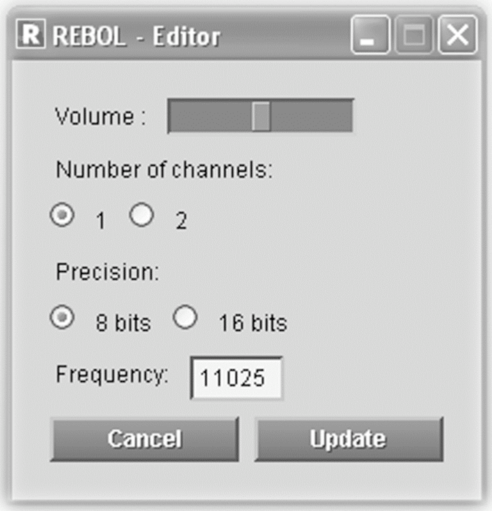


Figure 3-15. *The configuration box of the WAV player.*

The "Cancel" button allows the user to close the dialog box without changing the properties of the sound in memory. On the other hand, the "Update" button applies the changes made by the user to the sample.

Playing Samples

The user of our WAV sound reader clicks on the "Play" button to start playing sound samples. If the sample is loaded in memory, the data are inserted into the sound card port.

```
button "Start" [
      if not none? echantillon [
            port-sound: open sound://
            wait 0.1
            insert port-sound sample
            wait port-sound
            close port-sound
      ]
]
```

Once inserted in the port, the sample is immediately played. The optional use of the word `wait` tells the interpreter to wait until the end of the sound sample before continuing to execute the script.

Graphic display of a sound sample

Our WAV file player is already able to read and play a sound. To improve its look, we can display the sampled values as a real-time graph. The format of the data stored in the `data` property of the object holding the sample, is a model of simplicity. When the sample is an 8-bit one, each byte represents an amplitude. If the sound is stereo, it consists of two channels. Even bytes in the data, e.g.0 2 4 6 etc., are for the left channel. The sounds for the right channel are the odd numbered bytes, e.g.1 3 5 7 etc.. For sounds sampled with 16 bits, the logic is the same except each value is represented by two bytes.

We will use a `box` style and the `draw` dialect to visualise the different sound waves on the screen. To save time, the data is analysed when the WAV file is loaded. Two lists (`data-left` and `data-right`) are used to store the values extracted from the binary data in `echantillon/data`. First a `forskip` loops traverses the data. The inner `loop` is executed one or two times depending on the number of channels present in the file.

Generating a 16 bit value from two bytes is done by the simple operation: (byte1 * 256) + byte2 (multiplying a binary value by 256 has the same effect as shifting it eight bits to the left).

```
forskip son lg-package [
      canal: 1
      package: copy/part son lg-package
      loop echantillon/channels [
            either echantillon/bits = 8 [
                  valeur: first package
            ] [
                  valeur: ((first package) * 256) + (second package)
            ]
            either canal = 1 [
                  insert tail data-right valeur
                  canal: 2
            ] [ insert tail data-left valeur ]
            packet: skip packet next-channel
      ]
]
```

All that is left is for us to generate the graph using the Draw dialect. We will build the `trace-curve` function to perform this operation.

Two *refinements* (`/left` and `/right`) tell the function which channel is being displayed.

```
trace-curve: func [ data /left /right ] [
      either left [
            append canaux/effect/draw [ pen 255.0.0 ]
            depy: 99
            mult: -1
      ] [
            append canaux/effect/draw [ pen 0.255.0 ]
            depy: 101
            mult: 1
      ]
      data: at data (make integer! (sl-canaux/data / stp))
      repeat x 450 [
            xy1: make pair! reduce [ x depy ]
            finy: (100 + (mult * (make integer! ((first data) / stpy))))
            xy2: make pair! reduce [ x  finy ]
            append canaux/effect/draw reduce [ 'line xy1 xy2 ]
            data: next data
      ]
]
```

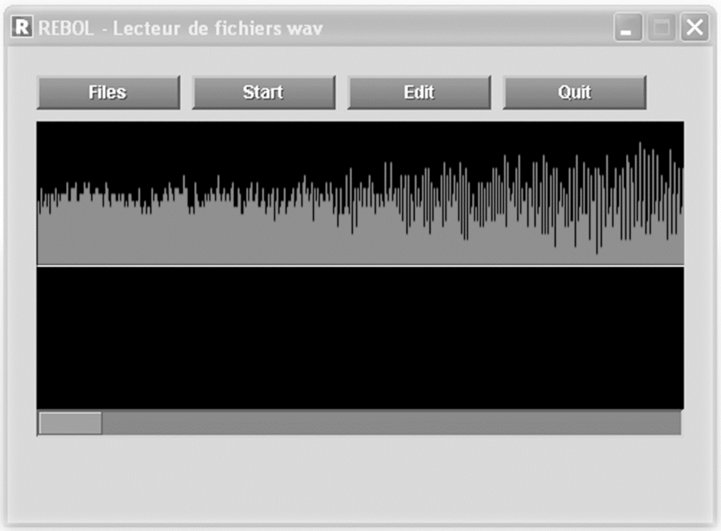


Figure 3-16. *Graphic display of a sound sample.*

The complete version of this sound player is less the 4KB of code and can be improved by the addition of new functionality such as cut and paste to perform digital editing or applying effects (echo, reverb, etc.). This is further proof of Rebol's simplicity and incredible versatility.

Summary

Rebol/View allows the easy development of platform-independent graphic user interfaces. The VID dialect can handle images, has many special effects and supports the simple redefinition of both the appearance and behaviour of graphic components. The sound options are limited but sufficient for many multimedia projects.

4 Networking and the Internet

Rebol is a messaging language intended for retrieving, manipulating and distributing data over networks. For this reason, it has many network programming tools.

Using TCP/IP protocols

Rebol is an excellent language for network programming. Without the need for a single extension, the Rebol interpreter is capable of using the main TCP/IP protocols. It also allows the definition of new network protocols. So it is not only possible to develop a client or server which uses an existing protocol but also one which uses a new protocol, adapted to your needs.

Protocols in Rebol

All versions of Rebol support ten common TCP/IP protocols. HTTP (*Hypertext Transport Protocol*) allows documents to be read from the web. Thanks to it, you can retrieve HTML documents, images, multi-media documents and even start the execution of cgi scripts on a distant sever. FTP (*File Transfer Protocol*) supports receiving and sending files with another network-connected computer. Electronic mail is managed with the support of the SMTP protocol for sending messages (*Simple Mail Transport Protocol*) and the IMAP (Internet Message Access Protocol) and POP3 (*Post Office Protocol*) protocols for receiving them. News on the Internet is read with the NNTP (*Network News Transmission Protocol*) protocol. There are also protocols for collecting information. It is possible to find the IP address of a server by interrogating the domain name server (DNS) and vice-versa. The "small" protocols which are Finger, Whois and Daytime make it possible to obtain a user's account information, administrative details of a domain name and the date and time from another machine.

Network configuration

Before doing anything at the network level with Rebol, you must first make sure your Rebol interpreter is correctly configured. If it's not, there is little chance that you will be able to reach the web or receive your electronic mail. If you didn't supply you network details to Rebol when you installed it, you must use the `set-net` word whose parameter is a block containing six values. You must put in this list:

- Your electronic mail address,
- The name or the IP address of the server for sending mail,
- The name or the IP address of the server for receiving mail,
- The name or IP address of a proxy,
- The port number of the proxy,
- The proxy type.

A proxy is an application which connects its clients to the internet. It is a method of releasing http requests to the internet and allowing the responses back to the computer which sent the request.

If you don't want to (you don't want to send email from Rebol) or don't need to (your computer is directly connected to the internet) supply a parameter, substitute it with the value `none`.

```
set-net [
      olivier.auverlot@domaine.fr
      smtp.domaine.fr
      pop3.domaine.fr
      proxy.domaine.fr
      8080
      generic
]
```

Rebol supports four different proxy types:

- `socks`,
- `socks4`,
- `socks5`,
- `generic`.

Sending and receiving email

You can now send and receive electronic mail. To do this you simply use the words `send` and `read`.

We can send a short message to a mail box with the command `send olivier.auverlot@domaine.fr "Cuckoo I am a message"`

If the body of the message contains a carriage return, you must enclose the character string between "{" and "}" which indicates a multi-line string:

```
send olivier.auverlot@domaine.fr {
      Cuckoo
      I am a message
}
```

By using the `header` refinement, you can add information such as a subject. The extra parameter for this refinement is an object whose various properties correspond to the many items that can be specified in an email header.

```
header: make system/standard/email [
      Subject: "message subject"
]
send/header olivier.auverlot@domaine.fr "hello"
```

Many internet services providers have recently started to use Extended Simple Mail Transfer Protocol (ESMTP) as a measure to improve security and combat spam. The current versions of Rebol include support for ESMTP through adding two new parameters to `set-net`; they are the account name and password:

```
set-net [
      olivier.auverlot@domaine.fr
      smtp.domaine.fr
      pop3.domaine.fr
      proxy.domaine.fr
      8080
      generic
      olivier
      homer
]
```

If you don't supply the account name and password, Rebol will request them when you try to send a message. (You should, of course, be very careful not to save your password in a Rebol source or text file).

To read your messages, all you need to do is use the word `read` followed by a URL consisting of the protocol (POP3 or IMAP), your account name, your password and the name of the mail server:

```
print read pop://olivier:homer@pop3.domaine.fr
```

If your email account name is your full email address including the @domain, it isn't possible to use this simple form to read mail. You can have to use the slightly longer, but more readable, method of defining a mailbox:

```
mailbox: [
      scheme: ‘pop
      host: “pop3.domaine.fr”
      user: “olivier.auverlot@domaine.fr”
      pass: “homer”
]

print read mailbox
```

Each message is received in the form of a multi-line character string which is place in a list. There are many ways to easily manage sending and receiving mails with Rebol. Any application can quickly be extended to exchange messages with people (alarms, automated reports,…) or other applications.

Accessing web resources

With the help of the HTTP and FTP protocols, you can read and send documents over a network. This lets us use Rebol as a navigator to retrieve HTML or XML files. This feature facilitates the construction of agents designed to search and harvest information. In fact, all you need to do is to specify a URL to read a resource on the network. The following example displays the contents of the homepage of Rebol Technologies:

```
print read http://www.rebol.com
```

By assigning the result of the request to a character string by using the word copy, you can carry out searches on it and extract information from it. With `load/markup`, the file received is analysed by the Rebol interpreter. You then will get a block composed of HTML or XML (`tag!`) tags and character strings.

With FTP, you can both send and receive files. The two words used are `read` and `write`. The URL passed as a parameter contains the user account and password needed to establish the connection.

With Rebol, the operations needed to manage distant files are just the same as those for local ones. In the example that follows, we will recover a file called `rapport.txt` from an FTP server and display it on the screen:

```
print read ftp://olivier:passwd@ftp.domaine.fr/docs/rapport.txt
```

And the other protocols?

With Rebol, all the protocols follow the same model; there are no special cases or exceptions. The complete set of protocols is based on a single syntax making then highly intuitive to use.

Suppose you want to find out the IP address of a machine. You simply use the syntax `ip-address: read dns://webserver`

Another example? You want to know the date and time of a server on the network. The syntax is evident:

```
print read daytime://myserver
```

Clients and servers

If there is one domain where Rebol is really impressive, it is probably the ease of developing client server applications using TCP or UDP network protocols.

To write a client, that is a program sending data to a server and waiting for a response, you first open a *socket* (the combination of an IP address and a port).

Then second, you insert the information to be transmitted to the server and then receive the answer. The last stage is to close the *socket* in order to release it.

```
REBOL [
      Subject: "client TCP"
]

; the script opens a connection to the machine at 172.29.143.1
; at port 8000/tcp
p: open tcp://172.29.143.1:8000
; a request (terminated by a carriage return) is sent
insert p "hello^/"
; the response is stored in a variable
response: copy ""
read-io p response 255
print response
close p
```

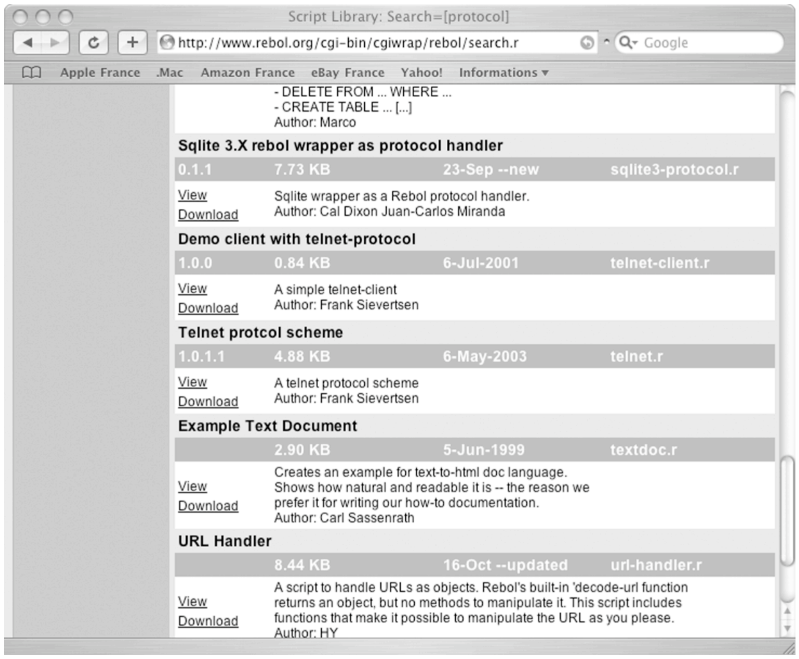
Writing a server doesn't pose any real problems. Just open a listening port and wait until some data is received. When data arrives, you process it and return a response to the client.

```
REBOL [
      Subject: "TCP server"
]
; open the port
p: open tcp://:8000
; the server enters an infinite loop
forever [
      wait p
      ; a connection is detected
      conn: first p
      ; read the request
      request: copy ""
      read-io conn request 255
      print [ "request ->" request ]
      ; send the response
      insert conn "OK^/"
      ; close the connection with the client
      close conn
]
```

Creating network protocols in Rebol

Designing network protocols is probably one of the most exciting aspects of the Rebol language. Thanks to them, you can interface you applications with TCP/IP, exchange data, create *peer-to-peer* communities or use web services.

By default, all Rebol versions contain a group of protocols covering a broad field of applications. With Rebol/Core and Rebol/View, you have ten network protocols ready for use (HTTP, POP, etc.). The commercial versions of Rebol provide other more specialised network protocols oriented towards *e-business*, in particular protocols for using the major DataBase Management Systems (DBMS), such as MySQL and Oracle, and those which provide secure data exchange (HTTPS). If however, you can't find what you want amongst these protocols, it is fully possible to create a new protocol that can be added to the existing ones that can be used in the same way as the standard ones. By checking out the different Rebol web sites, you will be able to find new protocols which you can extend (access to databases, telnet, SNMP, etc.).


Script Library: Search=[protocol]
http://www.rebol.org/cgi-bin/cgiwrap/rebol/search.r
Google
Apple France
.Mac
Amazon France
eBay France
Yahoo!
Informations ▾

- DELETE FROM ... WHERE ...
- CREATE TABLE ... [...]
Author: Marco

Sqlite 3.X rebol wrapper as protocol handler

0.1.1	7.73 KB	23-Sep --new	sqlite3-protocol.r
View Download	Sqlite wrapper as a Rebol protocol handler. Author: Cal Dixon Juan-Carlos Miranda		

Demo client with telnet-protocol

1.0.0	0.84 KB	6-Jul-2001	telnet-client.r
View Download	A simple telnet-client Author: Frank Sievertsen		

Telnet protcol scheme

1.0.1.1	4.88 KB	6-May-2003	telnet.r
View Download	A telnet protocol scheme Author: Frank Sievertsen		

Example Text Document

	2.90 KB	5-Jun-1999	textdoc.r
View Download	Creates an example for text-to-html doc language. Shows how natural and readable it is -- the reason we prefer it for writing our how-to documentation. Author: Carl Sassenrath		

URL Handler

	8.44 KB	16-Oct --updated	url-handler.r
View Download	A script to handle URLs as objects. Rebol's built-in 'decode-url function returns an object, but no methods to manipulate it. This script includes functions that make it possible to manipulate the URL as you please. Author: HY		

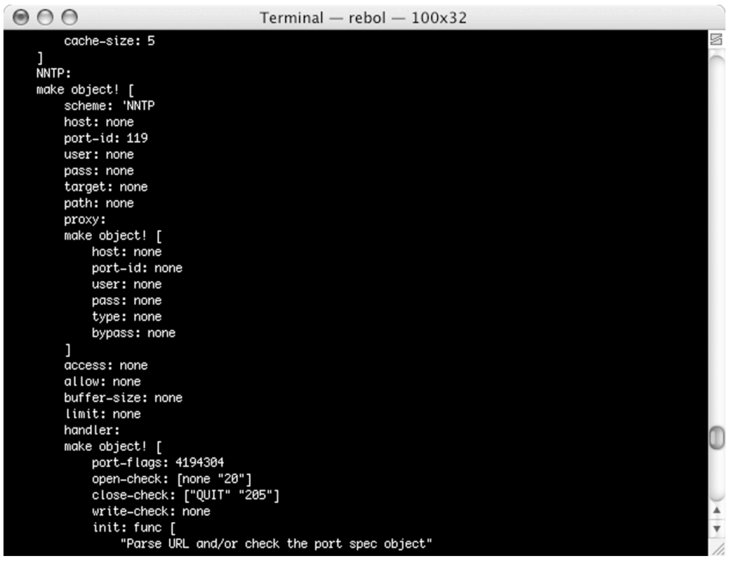
Figure 4-1. *Many protocols are available at www.rebol.org.*

You are probably thinking that this is really interesting but also very complicated, not at all! Rebol again demonstrates its excellent compromise between ease and power. Developing new network protocols is really within everybody's range.

The standard protocols

To understand the workings of network protocols in Rebol, it is best to study those included in the interpreter. Rebol is a meta-language, parts of Rebol are written in Rebol and this is the case with protocols. The different protocols are stored in the `schemes` property of the `system` object. To save them on to your hard disk, all you need to do is enter the following instructions into the console:

```
echo %protocols.txt
probe system/schemes
echo none
```



```
        cache-size: 5
    ]
    NNTP:
    make object! [
        scheme: 'NNTP
        host: none
        port-id: 119
        user: none
        pass: none
        target: none
        path: none
        proxy:
        make object! [
            host: none
            port-id: none
            user: none
            pass: none
            type: none
            bypass: none
        ]
        access: none
        allow: none
        buffer-size: none
        limit: none
        handler:
        make object! [
            port-flags: 4194304
            open-check: [none "20"]
            close-check: ["QUIT" "205"]
            write-check: none
            init: func [
                "Parse URL and/or check the port spec object"
```

Figure 4-2. *The protocols included in Rebol.*

You then get a textfile containing the source of all the protocols in Rebol. Each one of them is an object which inherits properties and methods from a root object, `root-protocol`.

The root protocol

To view the code of the `root-protocol`, you need to type the command `source root-protocol` in the console. This object, upon which all of Rebol's network protocols are based, contains the model for a fully functioning protocol.

It is already able to start a connection, insert data into and read data from a TCP or UDP port and to stop a connection to free up any resources used. In fact, all the programmer needs to do to make a new protocol is to decide whether it should use a TCP or UDP port. The most surprising example is probably the daytime protocol within Rebol. Its implementation takes only one line of code.

```
make Root-Protocol [ net-utils/net-install Daytime self 13 ]
```

This single line is sufficient to generate an object derived from `root-protocol`. The only modification that is absolutely necessary is, in fact, to the `net-util/net-install` method.

This one has three functions since it specifies the port used, the name of the protocol and, especially, inserts the new protocol into the property `system/schemes`.

Obviously, it is possible for you to develop more advanced protocols and in that case, it will probably be necessary for you to know and modify the different properties and methods contained in the `root-protocol` object. You will quickly see that each of them fulfils a very precise role.

Properties of the root-protocol object

The `root-protocol` object defines four properties which are `port-flag`, `open-check`, `close-check` and `write-check`. The first, `port-flag`, is probably the most difficult to understand since it allows you to specify the underlying communications protocol, TCP or UDP, to be used for exchanging data.

There are two modes for working with network protocols in Rebol: direct access or controlled access. With a direct access port (`system/standard/port-flags/direct`), the data are accessible character by character or line by line. Each time data is taken from the port it is immediately removed to allow access to the following data. This mode is particularly well suited to handling large volumes of information.

The controlled mode (`system/standard/port-flags/pass-thru`) gives greater freedom to the programmer in exchange for extra work. In effect, it is left to the developer to explicitly manage the receipt and storing of data. Information is generally held in a property of the protocol, so it is necessary to redefine all of the methods present so that they return all data collected to the program. The behaviour of ports can be further refined by applying a set of applicable constants using the binary operator `or`. So to set a port to work in controlled mode with binary data, it is enough to use the syntax:

```
port-flags: system/standard/port-flags/pass-thru or 32.
```

The properties `open-check`, `write-check` and `close-check` make it easy to automate the process when negotiating connection to a distant process, when writing data or closing a connection. The various stages of the negotiation are specified using a block containing the values to be transmitted and the data expected in return. Thus the block `[ "bye" [ 100 200 ] ]` tells the client to transmit the string "bye" and wait for the return values 100 or 200 before continuing. The sending and receipt of these values is not the programmer's responsibility since the methods of the root-protocol object deal with it all with the help of `net-utils/confirm`.

Methods of the root-protocol object

The `root-protocol` object contains 10 methods which take the port as their parameter. Each of them has a specific role in the working of the protocol.

The `init` method sets up the URL of the server to which the client must be connected. This method initialises the `user`, `pass`, `host`, `port-id`, `path` and `target` properties of the port object.

The `open-proto` method starts the connection to the server. This complex method deals with many operations such as proxy support. By default, Rebol's network protocols use TCP but is possible to opt for UDP by specifying the `/sub-protocol` refinement with the parameter, `'udp`, at the time this method is called.

`open-proto` is an important method but it isn't the true entry point of a protocol. In effect it is `open` which does that. By default, the `open` method actually calls `open-proto`. This default behaviour is seldom sufficient and you will often have to modify this method as intended by the designers.

The `close` method is called once the connection has ended and frees any resources used by the connection.

The `write` and `read` methods allow writing or reading data to or from a port and are invoked when a script uses the words `write-io` and `read-io`. The `get-sub-port` method returns the port used for communications between the client and the server. The `get-modes` and `set-modes` methods provide read and write access to the port's properties. Finally, the `awake` method is called when the receipt of data in the port is detected.

Implementing Echo

To better understand writing network protocols with Rebol, we are now going to study implementing the echo protocol. Defined by RFC 862, this service is very useful for finding out if a machine is connected to the network and even evaluating network performance. This protocol is quite straightforward; it is based on sending a character string from a client and receiving it back from the server. In Rebol, you will implement an echo protocol which sends the string "hello" to a distant server and waits for its response. Once this is received, the protocol client returns the time taken making it possible to monitor network performance.

The protocol will be added to the list of Rebol network protocols and will be usable by providing a URL specifying the protocol name (echo) and either the IP address or the domain name of the server.

The first step consists of generating an object inherited from `root-protocol` and setting the value of its properties. It would be a waste of time to modify `open-check`, `write-check` or `close-check` since no negotiation is needed to connect to the server with this protocol On the other hand, the values of `port-flags` of `system/standard/port-flags/pass-thru` must be set in order to work in controlled mode.

Which are the methods that need to be added or modified when the word `read` is applied to a protocol?

In fact, `read` calls three methods which are `open`, `copy` and `close`.

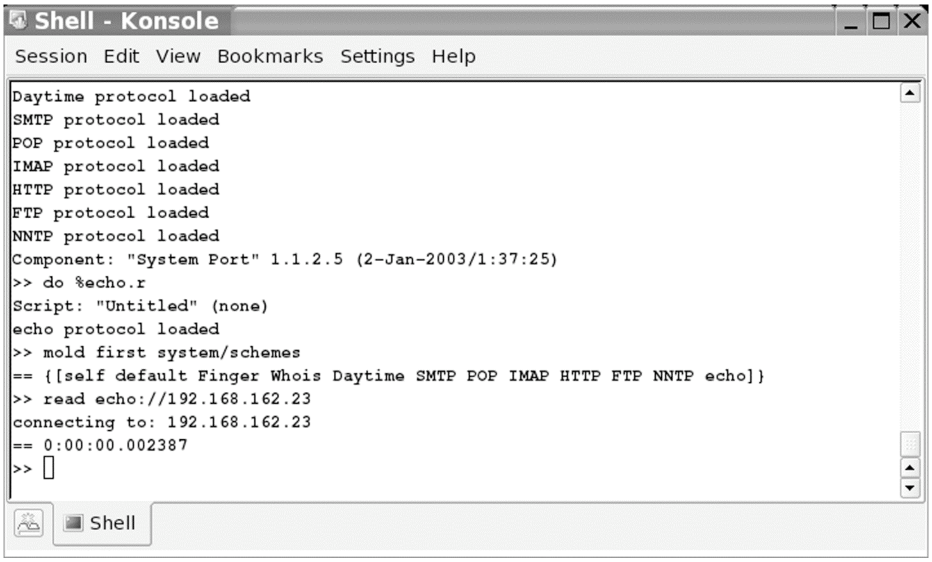
The `open` method calls `open-proto` and it isn't necessary to modify its behaviour for the echo protocol. The `close` shuts the TCP or UDP port used for the connection and also doesn't need to be modified. So you only need to add one method, `copy`, which has the job of sending and receiving character strings to and from the port. Be aware that you will redefine `copy` in the context of the protocol and that means using the word `copy` will cause the `copy` method you will have defined to be called and not the `copy` in the global context. Before modifying it, you must save the `copy` word by simply creating another word (`sys-copy`). Then the "new" `copy` doesn't pose any real problems.

The port to be used is passed as a parameter to the method and then all that is needed is to insert the character string and wait for it to be returned by the server. As there are no special end of string characters in this case, the `read-io` word is used to wait for the receipt of 5 characters. The method then finishes its work by returning the time taken for the whole process.

To add the echo protocol to the list of those known by Rebol, set the name of the protocol and, especially, specify the TCP port number used, it is necessary to complete the protocol declaration with the instruction `net-utils/net-install`.

```
echo-protocol: make root-protocol [
      port-flags: system/standard/port-flags/pass-thru
      sys-copy: get in system/words 'copy

      copy: func [ port /local reponse ] [
            t1: now/time/precise
            reponse: sys-copy ""
            insert port/sub-port "hello"
            read-io port/sub-port reponse 5
            (now/time/precise - t1)
      ]

      net-utils/net-install echo self 7
]
```

Once the protocol is loaded in the Rebol interpreter, simply entering `read echo://server-name` is enough to find out if the server is available and how long is the delay.



```
Daytime protocol loaded
SMTP protocol loaded
POP protocol loaded
IMAP protocol loaded
HTTP protocol loaded
FTP protocol loaded
NNTP protocol loaded
Component: "System Port" 1.1.2.5 (2-Jan-2003/1:37:25)
>> do %echo.r
Script: "Untitled" (none)
echo protocol loaded
>> mold first system/schemes
== {[self default Finger Whois Daytime SMTP POP IMAP HTTP FTP NNTP echo]}
>> read echo://192.168.162.23
connecting to: 192.168.162.23
== 0:00:00.002387
>>
```

Figure 4-3. *Using the echo protocol.*

Developing a gopher protocol

The gopher protocol was developed by Minnesota University. Defined in RFC 1436, it constitutes a distributed information system. It not only provides access to data (documents and files) but also to applications (telnet, 3270 terminal) through a tree structure. Based on the client/server model, the way gopher works is very simple. The client sends a request to TCP port 70 of the server. This indicates that the client wants to read a document on the server or a description of the contents of a directory. The end of the request is signified through the use of the CR and LF characters. In the case that the server returns the content of a file, the data is terminated by the CR and LF characters. For the description of the contents of a directory, each element is described in a character string terminated by a carriage return and the CR and LF characters mark the end of the response.

To use gopher with Rebol, the protocol client will provide two different methods. A programmer will be able to use `read` followed by a URL containing the name of the protocol (`gopher`), the name or IP address of the gopher server, and finally the path to the file requested.

The other method consists of opening a port using the gopher protocol with the word `open`, inserting the request in this port and then reading the answer before closing the port (`close`).

As with the `echo` protocol, you must use the `system/standard/port-flags/pass-thru` value for the `port-flag` property. The `path` property is intended for storing the client request. The `filetype` property contains the types of element recognised by gopher. So if the server returns that the element `test.txt` is type "0", the client protocol knows that it is a file.

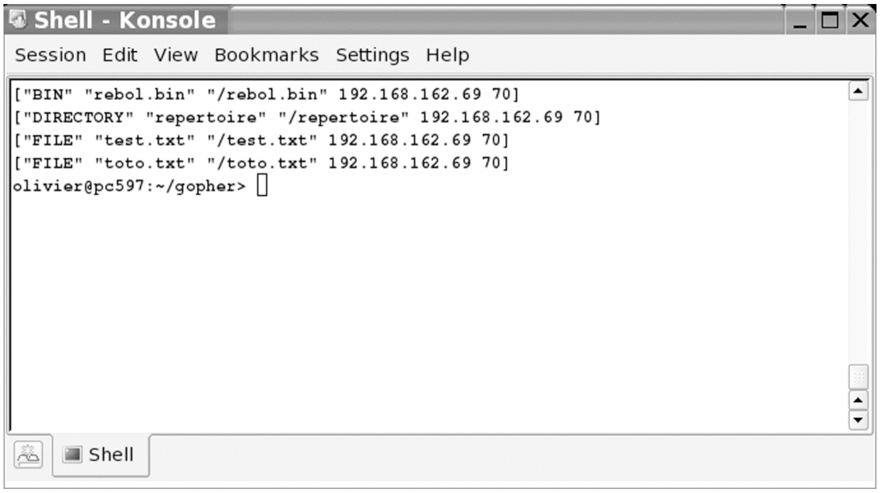
The methods `insert` and `copy` send both the two client requests via the `send-cmd` method. In effect, the `insert` method is called when you use the `insert` word in the global context, it is not if you use `read`.

In this case, only the `open`, `copy` and `close` methods are used; it is necessary to depend on `copy` to send the request. It just should be prevented from doing so twice. This is controlled through `send-cmd` and using the `/path` *refinement*. Once the command has been sent in the communications port (`port/sub-port`), the data are read by the `copy` method. If the last character of the client request is a "/", the data received by the client will be a description of the contents of a directory. The `list-directory` method handles the result. If the request was for a file, the contents of the response are returned by `copy`.

```
gopher-protocol: make root-protocol [

      port-flags: system/standard/port-flags/pass-thru

      sys-copy: get in system/words 'copy
      sys-insert: get in system/words 'insert
      sys-close: get in system/words 'close

      filepath: none

      filetype: [
            #"0" "FILE"
            #"1" "DIRECTORY"
            #"2" "CSOPHONEBOOK"
            #"3" "ERROR"
            #"4" "BINHEX"
            #"5" "BINDOS"
```

```
		#"6" "UUENCODE"
		#"7" "INDEX"
		#"8" "TELNET"
		#"9" "BIN"
		#"+" "REDUNDANTSERVER"
		#"T" "TERM3270"
		#"g" "GIF"
		#"I" "IMAGE"
		#"s" "AUDIO"
		#"M" "MIME"
		#";" "ANIMATION"
		#"h" "HTML"
	]

	send-cmd: func [ port /path data ] [
		if none? filepath  [
			either not path [
				either any [
					(none? port/path)
					(port/path = "/")
				] [
					filepath: sys-copy ""
				] [
					filepath: sys-copy port/path
					if not none? port/target [
						append filepath port/target
					]
				]
			] [ filepath: sys-copy data ]
			sys-insert port/sub-port filepath
		]
	]

	list-directory: func [ data /local content ] [
		content: sys-copy []
		foreach file data [
			file: parse file ""
			append/only content reduce [
				select filetype (first file/1)
				sys-copy/part (at file/1 2) ((length?
			file/1) - 1)
				sys-copy/part (at file/2 2) ((length?
			file/2) - 1)
				to-tuple file/3
				to-integer file/4
			]
		]
		content
	]

	insert: func [ port data ] [ send-cmd/path port data ]

	copy: func [ port /local item buffer result ] [
		send-cmd port
```

```
            buffer: sys-copy port/sub-port
            either (length? filepath) > 0 [
                  either (last filepath) = #"/" [
                        result: list-directory buffer
                  ] [ result: sys-copy buffer ]
            ] [  result: list-directory buffer ]
      ]

      close: func [ port ] [
            filepath: none
            sys-close port/sub-port
      ]

      net-utils/net-install gopher self 70
]
```



```
["BIN" "rebol.bin" "/rebol.bin" 192.168.162.69 70]
["DIRECTORY" "repertoire" "/repertoire" 192.168.162.69 70]
["FILE" "test.txt" "/test.txt" 192.168.162.69 70]
["FILE" "toto.txt" "/toto.txt" 192.168.162.69 70]
olivier@pc597:~/gopher>
```

Figure 4-4. *Using the gopher protocol.*

Rebol allows the development of network protocols in just a few tens of lines of code. This characteristic makes it an ideal language for network applications and information exchange. You can now expand your Rebol interpreter with many existing protocols and, why not, even invent some to meet your needs.

Rebol and CGI scripts

We now shift our attention to developing Web applications on an http server. With the help of Rebol/Core and Rebol/Command, you can design CGI scripts, programs that are run on an http server whose graphic user interface consists of HTML pages displayed in a browser.

The overall picture

A web application is a software program installed on an http server. The client machines connect to this application with a browser. The operating principle is based on the request/answer paradigm. The client queries the server with an http request and the server generally responds with an HTML page. What is great about this technology is that you don't have to install the application on the client desktop.

Everything is centralised on some machines which can easily be well cared for by the system administrators. Moreover, using an n-tier architecture (client, http server, database server) prevents the clients directly accessing the data. Only the HTTP server is configured for connecting with the DBMS. This solution provides a number of advantages for travelling users; especially those separated from their normal workstation.

If you want to develop a shopping mall or an information site on the Internet, CGI are an easy and quick way to develop a web application.

GCI overview

A user obtains information in a web page with the help of an HTML form. On clicking the submit button on a form (`submit`), the data in the form are transmitted to the CGI script specified in a URL. This program can receive the data in one of two methods:

- The GET method indicates that the parameters will be transmitted in the URL by the browser,
- The POST method signifies that the data are transmitted to the CGI separately.

The choice of one or another method depends on the context. The GET method limits the length of data transmitted to a URL and it is visible in the web browser. With POST, you are not limited by size and the data are hidden from the user. On the other hand, the extraction of the data by the CGI script is a little more difficult.

Once the parameters are received, the CGI script can do what it wants. Most often, this type of application connects to a database to save or retrieve information but nothing prevents you from sending emails or to reach files on the server. Your freedom depends on the rights granted to you by the machine's system administrator.

In all cases, the work of a CGI ends by generating a document whose format is determined by its MIME type: an HTML page to confirm the requested operation was successful or a PDF assembled by the Web server to allow the user to print it, etc.

How to write CGI scripts in Rebol?

First of all, you must configure your web server so that it allows the CGI scripts to be run. With Apache, it is necessary to review the ScriptAlias and AddHandler directives. You must also save your code in a directory authorised for CGI scripts. The main Web servers all have a `cgi-bin` directory dedicated to this task.

Let us start with something very simple, displaying a small text string in the browser's window. On a UNIX or Mac OS X system, you must use the first line of your script to associate your CGI script with the Rebol interpreter. The syntax is: `#!/usr/bin/rebol –cs`. The `-c` option forces the interpreter to run in CGI mode. As for the `-s` option, it turns Rebol file security off. (If you are using Apache on a Windows system, you can use a Windows specific version `#!c:/rebol/rebol.exe –cs).`

You must then specify the type of document (`Content-Type`) intended for the client browser. This is part of the http header and is separated from the body of the document by two carriage returns. What follows it is the Rebol code to be run.

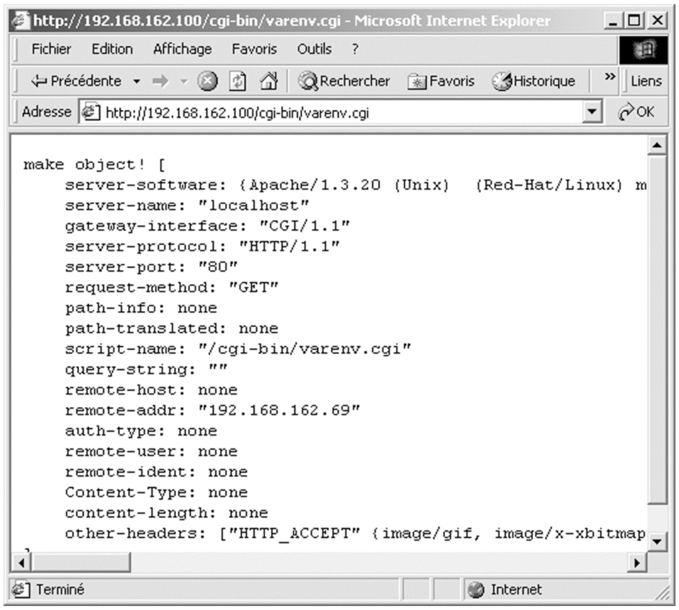
```
#!/usr/bin/rebol -cs

REBOL [ ]
print "Content-Type: text/plain^/"
print "Hello !"
```

This file, called `test.cgi`, is marked as an executable file on Unix and Mac OS X systems with the command `chmod +x test.cgi`. All that is left is for you to test the script with your browser. The URL should look something like this `http://<server_name>/cgi-bin/test.cgi`.

Reading parameters

Suppose you now want to send the contents of an HTML form to your CGI script. This poses the problem of reading the information sent by the browser. For that you have a group of environment variables accessible with the help of Rebol's `system/options/cgi` object. Each of its properties maps to one of the http server's environment variables. The only exception is the `other-headers` property which allows the use of various characteristics of the HTTP protocol such as cookies. The following CGI script displays all of the variables in your browser:

```
REBOL [ ]
print "Content-Type: text/plain^/"
print mold system/options/cgi
```


http://192.168.162.100/cgi-bin/varenv.cgi - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente
Rechercher
Favoris
Historique
Liens
Adresse http://192.168.162.100/cgi-bin/varenv.cgi
OK
Terminé
Internet

```
make object! [
    server-software: {Apache/1.3.20 (Unix)  (Red-Hat/Linux) m
    server-name: "localhost"
    gateway-interface: "CGI/1.1"
    server-protocol: "HTTP/1.1"
    server-port: "80"
    request-method: "GET"
    path-info: none
    path-translated: none
    script-name: "/cgi-bin/varenv.cgi"
    query-string: ""
    remote-host: none
    remote-addr: "192.168.162.69"
    auth-type: none
    remote-user: none
    remote-ident: none
    Content-Type: none
    content-length: none
    other-headers: ["HTTP_ACCEPT" {image/gif, image/x-xbitmap
```

Figure 4-5. *CGI environment variables.*

To understand the use of these variables, you will build a small web application in which the user enters his name. The server responds with a greeting page. For example, if the user types "Olivier" and "Auverlot", the server generates a page containing "Hello Olivier Auverlot". You must first, design the HTML form to capture the user’s name and save it in a directory in the tree structure of your HTTP server.

```
<html>
<body>
<form name="myform" method="get"
action="http://server.domaine.org/cgi-bin/hello.cgi">
<input name="firstname" type="field" value=""><br>
<input name="name" type="field" value=""><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

This HTML page defines a simple form and once the user clicks the “Submit’ button, the contents are transmitted to the `hello.cgi` script using the GET method.

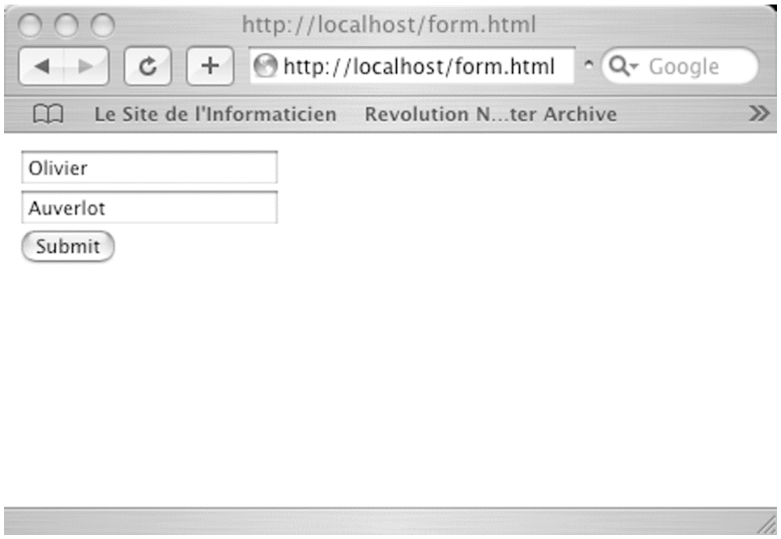
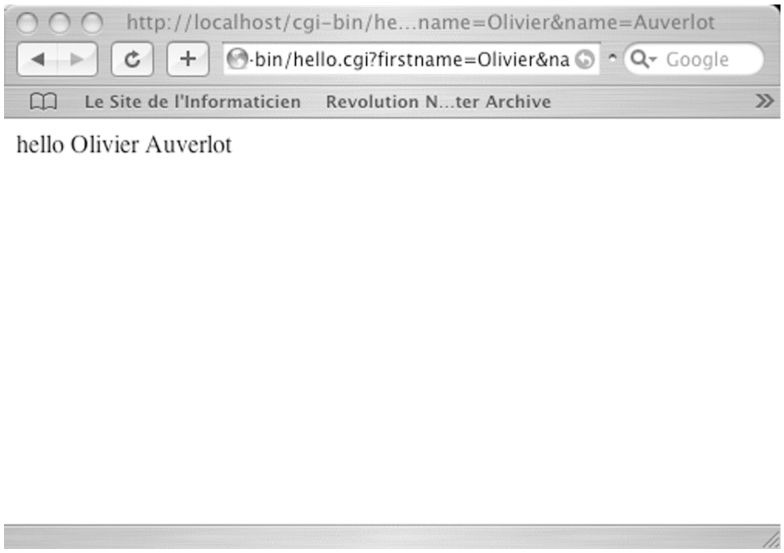

http://localhost/form.html
http://localhost/form.html
Google
Le Site de l'Informaticien
Revolution N...ter Archive
Olivier
Auverlot
Submit

Figure 4-6. *Submitting data to an http server.*

On the server, the script must generate an HTML page based on the contents of the `QUERY-STRING` variable. The data transmitted by the browser are in MIME format and are not directly usable. In this example, the `QUERY-STRING` variable contains "firstname=olivier&name=auverlot".

To ease your workload, Rebol contains the word `decode-cgi` which creates an object from a MIME format character string: each of its properties corresponds to a field in the form.

```
rebol [ ]
print "content-type: text/html^/"
info: make object! decode-cgi system/options/cgi/query-string
print "<html><body>"
print [ "hello " info/firstname info/name ]
print "</body></html>"
```

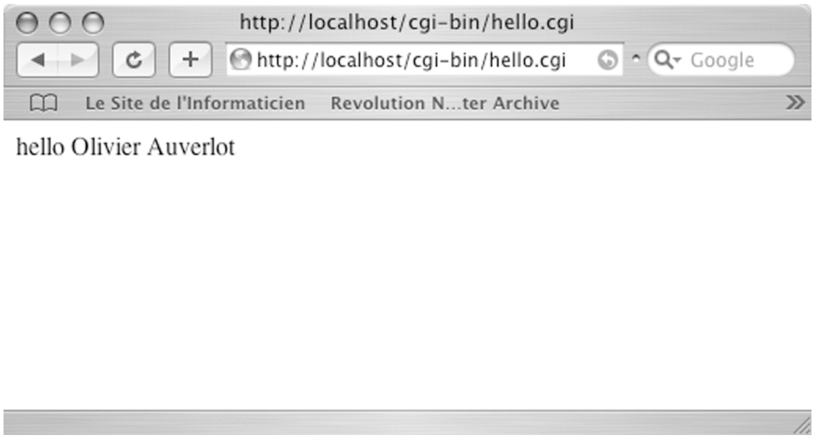

http://localhost/cgi-bin/he...name=Olivier&name=Auverlot
-bin/hello.cgi?firstname=Olivier&na
Google
Le Site de l'Informaticien
Revolution N...ter Archive

hello Olivier Auverlot

Figure 4-7. *The output of the CGI script.*

By using the `REQUEST-METHOD` variable, your CGI script can determine the method of data transmission used by the client machine. Simply check the contents of this variable to automatically adapt your script to one or other of the two methods. If the content of the form is transmitted using the POST method, the CGI script must also read the number of bytes specified in the `CONTENT-LENGTH` property from the standard input file.

```
rebol [ ]
print "content-type: text/html^/"
print "<html><body>"
either system/options/cgi/request-method = "get" [
      data: system/options/cgi/query-string
] [
      data: copy ""
      length: to-integer system/options/cgi/content-length
      until [
            buffer: copy ""
            read-io system/ports/input buffer (to-integer
      system/options/cgi/content-length)
            append data buffer
            ((length? data) = length)
      ]
]
info: make object! decode-cgi data
print "<html><body>"
print [ "hello " info/firstname info/name ]
print "</body></html>"
```


http://localhost/cgi-bin/hello.cgi
Google
Le Site de l'Informaticien
Revolution N...ter Archive

hello Olivier Auverlot

Figure 4-8. *The transmitted with the POST method is not visible in the URL.*

Writing CGI scripts in Rebol is not at all difficult. It is a simple, robust technology with which web applications can be rapidly developed.

Producing dynamic web documents

CGI aren't the only way to execute Rebol code on a web server. Magic!, an extension to the Apache web server written in Rebol, lets you place Rebol instructions inside your HTML pages. Just like PHP, JSP and ASP, Magic! executes Rebol code on the web server. This is totally transparent to the client and enables the very quick development of interactive websites. The main advantage of this solution is that it isn't necessary to set the rights of each user to execute CGI scripts.

Dynamic pages designed with Magic! are just the same as other documents to the web server and are stored in the same directory as static html pages.

How Magic! works

Magic! takes the form of an extension to the Apache HTTP server. Being written in Rebol, it works on all the platforms supported by Rebol/Core and also can use the extra features of View/Pro and Rebol/Command if one of them is present. Magic! is, in fact, a simple, short CGI that contains a complete set of functions to take all of the hard work out of developing dynamic pages. The mechanism it uses is very simple: when the Apache server encounters a `*.rhtml` page, it executes the magic.cgi script which analyses the document and executes and Rebol code found inside it. The final result is transmitted to the client which only sees a simple HTML page. Using Magic! you are not restricted to just HTML pages, you can also produce documents in other formats, such as an XML document, an image or even a pdf.

Installation

Magic! and its French documentation are available from http://www.auverlot.fr/Fichiers.html. The `magic.cgi` script needs to be placed in the `cgi-bin` of your Apache server with the correct execution rights. You may also need to change the shebang line (first line) in the `magic.cgi` file so that it correctly describes the file path to your Rebol interpreter.

The rest of the work to install Magic! consists of configuring Apache so that it can manage its newfound capabilities. For this, you must modify Apache's `httpd.conf` file. The first stage consists of authorising the use of CGI scripts in order to make `magic.cgi` available for work.

So you must declare the default directory for CGI scripts and signify the file extension to be used by them:

```
ScriptAlias /cgi-bin/ "/Library/WebServer/CGI-Executables/"
AddHandler cgi-script .cgi
```

After that, you can now modify the `httpd.conf` file so that it recognises the Magic! file extension (.rhtml). This is done with the `Addhandler` directive, you assign the "magic" event to files with the rhtml extension and through the Action directive you tell Apache that all files raising the "magic" event must be re-directed to the `magic.cgi` script.

```
AddHandler magic .rhtml
Action magic /cgi-bin/magic.cgi
```

You could also change the setting of the `DirectoryIndex` directive in order to add the index.rhmtl page to the sites default pages. This simple change enables you to set up a dynamic home page.

```
 DirectoryIndex index.rhtml index.html
```

All that is left to do now is to re-start the Apache server to apply the changes you've just made to the configuration. To do this, Linux RedHat users should use the `service httpd restart` command. On other systems (MacOS X, BSD or other Linux distributions), you use `apachectl restart`.

When the static becomes dynamic

To test the Magic! installation, the most effective method is to type your first dynamic paged called `test.rhtml`. This looks like a normal HTML page except that the document contains two new tags: `<rebol>` and `</rebol>`. Between these tags, a simple line of Rebol code that displays the current time.

```
<html>
<head></head>
<body>
<rebol>
     print now/time
</rebol>
</body>
</html>
```

Obviously, you can insert as many `<rebol></rebol>` blocks as you want in a RHTML page.

Rebol code can also be placed at the beginning and end of the document. The following example demonstrates that it is easy to separate the definition of a function from its use:

```
<rebol>
      current-time: does [ print now/time ]
</rebol>
<html>
<head></head>
<body>
      <rebol> current-time </rebol>
</body>
</html>
```

It is common in web applications to transmit the data entered in a form to a CGI script or dynamic page. Magic! supports this operation with the help of a dedicated object named `vars`. Each property in this object corresponds to a field in the transmitted form. Also Magic! automatically detects the data transmission method used by the form (GET or POST). For the programmer, this means reading the data is totally transparent. The following example consists of an HTML form in which the name and first name of the user are entered. The data are transmitted to the page `post.rhtml` using the POST method:

```
<html>
<head></head>
<body>
<form name="info" method="post" action="post.rhtml">
      Name:<input type="field" name="name"><br>
      First Name:<input type="field" name="first-name"><br>
      <input type="submit" value="Submit">
</form>
</body>
</html>
```

The destination page can then display the data received and accessible through the `vars` object:

```
<html>
<head></head>
<body>
<rebol>
      print [ "hello " vars/first-name " " vars/name ]
</rebol>
</body>
</html>
```

As you can already see, Magic! enormously facilitates the work of the programmer but its possibilities don't stop there. It provides a number of other functions dedicated to dynamic page support.

Magic! functionality

Magic! is a genuine toolbox for web developers. It actually adds a set of new words to the Rebol dictionary allowing shared code libraries, protecting the web server by controlling user's access, modifying the MIME type of generated documents, setting and reading cookies and, finally, managing sessions.

Library functions

One of Magic!'s most helpful functions is probably its `library` word which allows Rebol code sections to be stored in separate scripts. With it the duplication of code in different files can be avoided, so it facilitates the sharing of software components between developers. To implement this function, all that is needed is to create a library directory and inform Magic! of its access path by setting the `m-library-path` variable in the `magic.cgi` file. So if you want all developers of a Magic! based system to have access to SoftInnov's free MySQL protocol, you simply save a copy of `mysql-protocol.r` in the library directory. The following example is a RHTML page using that protocol to access a database.

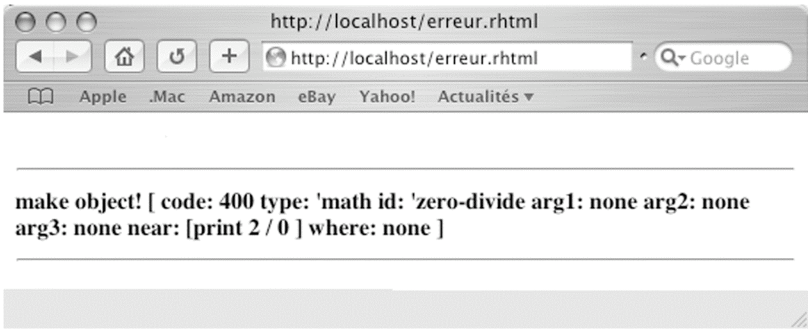
```
<html>
 <head></head>
 <body>
     <rebol>
         library %mysql-protocol.r
         db: open mysql://olivier:homer@localhost/mysql
         insert db {select * from user}
         print mold copy db
         close db
     </rebol>
 </body>
 </html>
```

Controlling embedded Rebol code

Security is an essential characteristic of Magic!. It is very important to control programmer's actions so that cannot compromise server stability or run mavolent scripts which read or delete data which doesn't belong to them. To resolve these problems, dynamic pages designed with Magic! conform to a strict security policy by default:

- dictionary words cannot be redefined,
- script can read files in the library directory,
- scripts can read and write files in the current directory,
- all other directories and files in the file system hierarchy are inaccessible.

This basic policy is contained in the `magic.cgi` file and can easily be adapted to your specific needs. You should also know that any Rebol code contained in a Magic! page is closely supervised to intercept any possible execution errors.



```
make object! [ code: 400 type: 'math id: 'zero-divide arg1: none arg2: none
arg3: none near: [print 2 / 0 ] where: none ]
```

Figure 4-9. *An execution error handled by Magic!.*

If there is a problem the evaluation of the Rebol code is immediately stopped and an error message returned for the browser to display.

Handling MIME types

If you want to generate anything other than an HTML page, you should select a different MIME type by modifying the contents of the HTTP header returned to the client. Thanks to this option, you can produce XML, ASCII data, an image, or even a PDF document.

To specify the MIME type, you use the word `header` followed by a character string containing the HTTP content type you want to use.

The following example shows how easy it is to produce a PDF document with the help of Gabriele Santilli's `pdf-maker` library (available from his rebsite http://web.tiscalinet.it/rebol/index.r). To test it, don't forget to save a copy of the `pdf-maker.r` file in the Magic! library directory.

As the generated file is a PDF document, the content of the dynamic page is a single <rebol> tag as HTML tags are of no use.

```
<rebol>
      library %pdf-maker.r
      header "Content-type: application/pdf"
      print to-string layout-pdf [
            [
            textbox ["I am a PDF document"]
            circle 50 250 20
            ]
      ]
 </rebol>
```

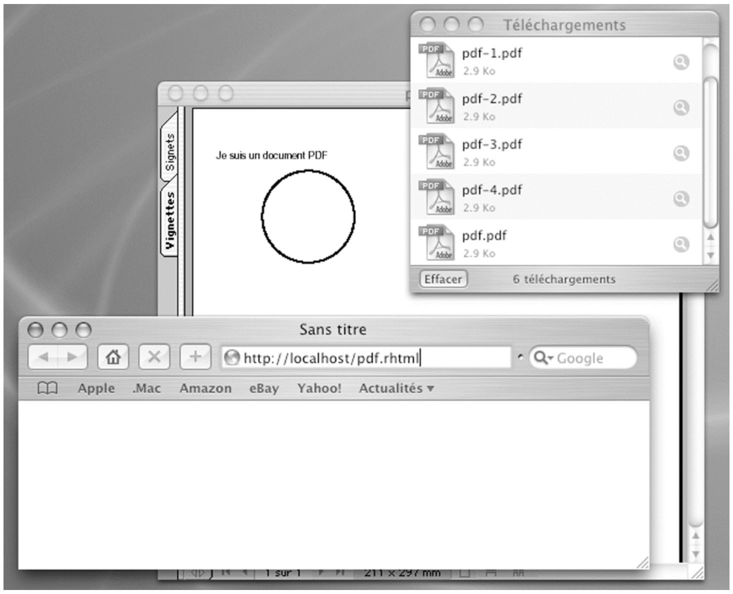


Figure 4-10. *Generating PDF documents.*

Managing cookies

Cookies can be managed with the `setcookie` and `getcookie` words. They are intended for sending and receiving cookies respectively. These two words can be used at any point in a RHTML page and directly modify the http header generated by Magic!. `setcookie` uses two mandatory parameters which are the name of the cookie and its value. You can precisely specify `setcookie`'s behaviour through *refinements* such as `/expire` which sets the length of a cookies' validity, `/domain` and `/path` which indicate the field to which the cookie is attached and an access path, `/secure` in order to avoid sending the cookie to an insecure server. `getcookie` is much easier to use as it only takes one parameter, the name of the cookie for which you want the value.

The following example is a RHTML page which generates a cookie called `test`, setting the creation time to the time of the session and allowing access from all URLs of the origination server. The value of this cookie is modified by assigning a random number. With the help of `getcookie`, the current content of the cookie is retrieved and displayed by the RHTML page.

```
<html>
<head></head>
<body>
 <rebol>
     random/seed now/time/precise
     v: random 100
     setcookie/path "test" v "/"
 </rebol>
 The value of the cookie is:<rebol>print getcookie "test"</rebol>
 </body>
 </html>
```

Managing sessions

Magic! is a considerable help with one important aspect of writing web applications in Rebol by providing support for sessions. The objective of these is to make it possible for a program to identify a user and store data that can be used in various pages of the web application that they access.

This mechanism serves to compensate for an architectural choice made by the originators of the http protocol: the fact that HTTP is “stateless” and web servers forget every request once they have responded to them. One can sum up the situation in a single, simple sentence: a web application user does not exist between two pages. The idea of sessions consists of allocating an identifier to a user and to recover it in one way or another with each of their requests. Once the user has a reference and is known, it becomes possible to store data on the server and simplify the design of the application by avoiding the use of hidden variables (fields hidden inside HTML forms) and other similar tricks.

Magic! stores session variables in an object stored in the directory referred to by the `m-sessions-path` word in `magic.cgi`. A unique object is created for each user; the filename is assigned a unique name: the `MAGICSESSID`.

To create a session, you use the `session-start` word which generates a session identifier. Using the `/cookie` *refinement*, the `MAGICSESSID` will be automatically transmitted at each interaction between the client and the server via a cookie.

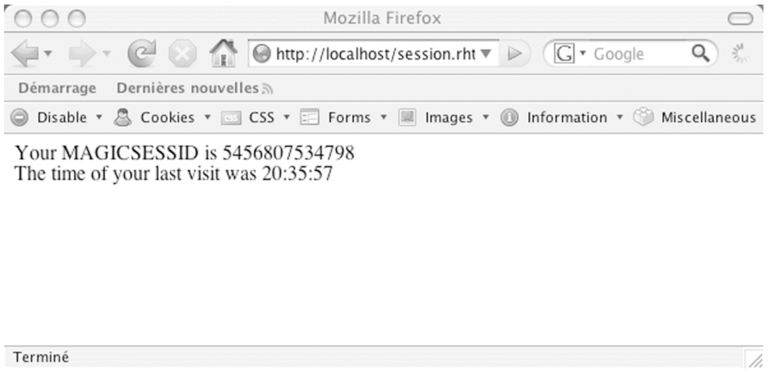
By using the *refinement* `/expire`, you can specify the time allotted to the session, overriding the default value. This determines the length of time without user activity before the session is considered inactive and can be destroyed. You can force a session to be cancelled can with the word `session-destroy`. To verify that a session has been started, you can use the word `session?` which returns a boolean value.

With the `session-vars` word, you can create session variables and assign an initial value to them. Once generated, these variables are accessible in different pages of your application with the help of a dedicated object called `session` each property of which corresponds to a session variable.

In the example that follows, the dynamic page checks if a session is already active and, if not, starts a new session which will end after ten minutes of inactivity. The MAGICSESSID of the new session is automatically transmitted to the cookie.

A session variable, `last-visit` is also declared. The script then displays the user identity and value of the `last-visit` which contains the time the user last visited the page. The current time is then saved in the session variable.

```
<html>
 <head></head>
 <body>
     <rebol>
         if not session? [
             session-start/cookie/expire 0:10:00
             session-vars [ [ last-visit: "" ] ]
         ]
         print [ "Your MAGICSESSID is " MAGICSESSID "<br>" ]
         print [ "The time of your last visit was " session/last-visit ]
         session/last-visit: now/time
     </rebol>
 </body>
 </html>
```



```
Your MAGICSESSID is 5456807534798
The time of your last visit was 20:35:57
```

Figure 4-11. *The time of the last visit is saved in the session variable.*

That is all there is to creating dynamic web pages with Rebol and Magic!. The combination facilitates the quick and easy development of web applications as it relieves the programmer from a great number of the troublesome details of doing so.

Handling XML documents

Do you really need XML when you're using Rebol? The answer to this question is not as obvious as it first seems. Indeed, on paper, Rebol and XML are more competitors than partners as they each are based on their own universal format for organising and storing data.

The problem of interoperability

Rebol is a meta language and a Rebol script initially is just a data structure whose elements become instructions only at the time of evaluation. That means that Rebol's syntax makes it possible to write scripts, databases or configuration files. The block format is perfectly adapted to storing data and transporting it via the TCP/IP protocols. The following example illustrates the use of Rebol as a format for storing information:

```
[
      expiry-date: 15/1/2003
      filename: %journal.txt
      users: [
            [ "Olivier" category: 3 ]
      ]
]
```

If Rebol and XML play in the same field, what is the interest in making them work together?

The problem is simply that not everyone uses Rebol to develop their information systems. An organisation can of course choose IOS as a server, develop web applications with Rebol/Command and use Rebol native format blocks for storing and transporting data. The problem arises when it wishes to communicate with another organisation whose business applications use other technologies. In this case, XML is quite simply the only solution.

Rebol is closed and includes only the functionality necessary to exchange data between its different versions. However choosing Rebol does not cut you off from the rest of the world. Rebol understands and speaks XML, the inverse is not true…

Rebol's integrated parser

The Rebol interpreter includes a parser which can transform an XML document into a group of easily usable blocks. The XML document is also translated into Rebol native formats to optimise its use. In fact, you only need to know one word: `parse-xml`. It takes a single parameter of a string of data formatted according to the XML specifications.

The parser is of the non-validating type. This means that it checks the syntax of the document but not the validity of the data contained in it. If a DTD (*Document Type Definition*) is specified in the XMLcode, it will simply be ignored.

Let's start by creating an XML file called `contacts.xml` with the help of a simple text editor. This document contains a list of people identified by their name and first name.

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<contacts>
      <person sex="female">
            <name>Dziubak</name>
            <firstname>Nathalie</firstname>
      </person>
      <person sex="male">
            <name>Auverlot</name>
            <firstname>Olivier</firstname>
      </person>
</contacts>
```

To transmit this file to the Rebol interpreter, all you need to type is:

```
parse-xml read %contacts.xml
```

The word `read` reads the XML file and returns the contents as a character string which is passed to Rebol's XML parser. The XML data is transcribed to Rebol blocks which are displayed on the screen.

How to use the data?

Before extracting the information from these blocks, you must assign the result of the XML parser to a word. Once this operation is finished, you can simply use the standard Rebol words for handling data lists. Contrary to other languages, Rebol does not have libraries such as DOM or SAX for the simple reason that they are not of great use to Rebol. In effect, once the parser has analysed a file, Rebol does not handle the XML any longer, just the resulting Rebol data structures.

In the memory of the Rebol interpreter, the data of the `contacts.xml` file are stored in the following way:

```
[document none [["contacts" none ["^/^-" ["person" ["sex" "female"]
["^/^-^-" ["name" none ["Dziubak"]] "^/^-^-" ["firstname" none
["Nathalie"]] "^/^-"]] "^/^-" ["person" ["sex" "male"] ["^/^-^-" ["name"
none ["Auverlot"]] "^/^-^-" ["firstname" none ["Olivier"]] "^/^-"]]
"^/"]]]]
```

At the first glance, one feels a little depressed by this display of a tangle of blocks and carriage returns indicated by the "^/" control characters.

Don't worry, it is not very difficult to understand and, especially, this data organisation is very simple to handle in Rebol. In fact, this list of blocks is composed according to the following rules:

- each XML entity is represented by a list composed of three values organised in the model [element attributes content],
- the word `document` corresponds to the root of the XML tree,
- attributes are presented in the form of attribute name – value pairs,
- if one of more attributes are present, they are integrated in the list,
- the absence of attributes in an XML tag is indicated by the value `none`,
- the carriage return characters ("^/") are intended to improve the display of the data when the `print` word is used. You can simply ignore them during analysis of the document tree structure.

From these rules, you can write a script to access the data in the contacts.xml file. This program displays the name and first name of each person indicated. For each person, the code displays a message showing the value of the attribute from the `<person>` tag indicating their sex.

The script uses a recursive procedure, `analyse`, which takes the block to be traversed as a parameter. The block is assigned the word `data`, from which you can deduce that the first, second and third elements correspond to the XML tag, to the list of attributes and the value of the contents of the tag. With the help of simple tests, you can also display the data extracted for the XML document. It is also possible to assign the results to words for later use.

```
REBOL [
        subject: "Traverse contacts.xml"
]

analyse: function [ data ] [ value ] [
      if block? data [
            if data/1 = "person" [
                  print [ "New person" data/2 ]
            ]
            if any [
                  data/1 = "name"
                  data/1 = "firstname"
            ] [ print data/3 ]
            if not none? data/3 [
                  value: data/3
                  if (length? value) > 1 [
                        forall value [
                              if block? value [
                                    analyse first value
                              ]
                        ]
                  ]
            ]
      ]
]
```

To use the `analyse` function, all you need to do is call it with a parameter of a block containing the data from the XML file. This can be done with a simple line of code:

```
analyse first third parse-xml read %contacts.xml
```

The guts of the parser

The Rebol XML is itself written in Rebol. The command `source parse-xml`, entered into the Rebol console, will show you that calling the word `parse-xml` actually calls the `parse-xml` method of an object called `xml-language`. You can then use the command `source xml-language` to discover the contents of this object which contains the properties and methods of the XML parser.

Note that you have the possibility to modify behaviour of the XML parser to adapt it to your own needs. The properties, methods and rules can indeed be redefined as you care.

Also, the `xml-language/verbose` property which contains a boolean value, permits the selection (or not) of a mode in which the XML parser displays its activity as it parses the XML data. The `xml-language/check-version` method is also very interesting since its deactivation makes it possible to avoid displaying the version number specified in the XML file header.

```
xml-language/verbose: false
xml-language/check-version: false
```

A better XML parser

As the code of the XML parser is accessible, some programmers have tried to improve its functionality. This is the case with Gavin F.McKenzie who developed an improved version which can be found via the Internet (http://www.rebol.org). His version supports `cdata` sections indicating tags to be ignored during analysis and can recognise the elements of an internal DTD.

Before testing his version, you must modify the `contacts.xml` document in order to add new elements such as an internal DTD and a CDATA section.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE contacts [
<!ELEMENT contacts (person+)>
<!ELEMENT person (nane,firstname)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ATTLIST person sex (male|female) #REQUIRED>
]>
<contacts>
        <person sex="female">
                <name>Dziubak</name>
                <firstname>Nathalie</firstname>
        </person>
        <![CDATA[ TEXT NOT USED BY THE XML PARSER]]>
        <person sex="male">
                <name>Auverlot</name>
                <firstname>Olivier</firstname>
        </person>
</contacts>
```

To use Gavin F. McKenzie's XML parser, you must include the library script `xml-parse.r` in your script by using `do`.

Then all that you need to do is use `parse-xml+` against the XML file, simply using the command `parse-xml+ read %contacts.xml`.

Compared to the standard Rebol XML parser, parse-xml+ provides more information about the document and its analysis is more precise. The DTD is stored in the element `subset` of the `document` block. The `cdata` sections become simple comments.

For extra information, Gavin F. McKenzie proposed an original solution which consists of converting an XML document XML into a tree structure made up of objects. The library script is called `xml-object.r` which must be included in your scripts with `do`. The following instructions transform your `contacts.xml` document into a group of objects in the context of the `obj-xml` object:

```
obj-xml: make object! xml-to-object parse-xml+ read %contacts.xml
```

The root element of the XML tree corresponds to the `document` object, you can deduce the XML version from the `obj-xml/document/version` property and the DTD can be obtained via the `obj-xml/document/subset` property. For each of the sub-objects, the property `value?` lets you know the value of the XML element. So it becomes very simple to extract the names of all the people in the list of contacts:

```
foreach person obj-xml/document/contacts/person [
      print person/name/value?
]
```

If an element contains a combination of sub-elements and text, the `content?` property lets you retrieve the different values in a block. The XML tags are represented by words and the information as character strings. To test this property, you can extract the contents of the element `<contacts>` in this way:

```
print mold obj-xml/document/contacts/content?
```

This object approach to XML documents makes it very easy for the programmer wanting to extract information from it.

Generating XML

Using XML with Rebol is not restricted to just reading files. In numerous situations, it is necessary to generate XML documents from Rebol data structures. Whilst this functionality isn't part of the basic dictionary, it isn't difficult to implement.

Suppose that you work for a company which has its spare parts catalogue structured in Rebol blocks. Ideally these items should be transformed using a to-xml function in the following way:

```
to-xml [
      catalogue [
            part [ ref “243a” name “nut” ]
            part [ ref “784c” name “bolt” ]
      ]
]
```

We will write this function. Progressively through its construction, the XML document will be saved in the variable `docxml`. The function traverses the data block two elements at time. The first value always corresponds to an XML element and causes the generation of an XML opening tag with the `to-tag` word. If the second value is a character string, the data is appended to `docxml` and a closing tag is added. On the other hand, if a block is encountered, the `to-xml` function is called again in a recursive fashion to traverse the whole data tree.

```
docxml: copy {<?xml version="1.0" encoding="iso-8891-1" ?>^/}

to-xml: function [ blk ] [ ] [
        forskip blk 2 [
                append docxml to-tag first blk
                either block? second blk [
                        to-xml second blk
                ] [
                        append docxml form second blk
                ]
                append docxml to-tag join "/" first blk
        ]
        docxml
]
```

As you can see, it is extremely easy to import and export XML formatted data with Rebol. Within the framework of internet applications, the use of XML is of obvious interest in many cases. You will not have any difficulty in generating XML bound for a web browser or even remote procedures using protocols such as SOAP or XML/RPC.

Using Web Services

We are now in the age of *Web Services*, that is the remote use of software distributed across many servers. There are several methods to use such an architecture. If the available SOAP support for Rebol at (http://compkarori.com/soap) is not yet totally comprehensive, Maarten Koopmans's Rugby is an effective broker, the latest version is available at http://www.hmkdesign.dk/rebol/page0/page0.html and a very powerful XML-RPC library called RebXR has been developed by Andreas Bolka (http://earl.strain.at). Rebol Technologies has recently released an experimental version of Rebol/Services specifically designed for developing web services in Rebol.

Introducing XML-RPC

XML-RPC (*Remote Procedure Call*) is a communications protocol resulting from the work of Dave Winer. It uses an HTTP like transport layer and XML for encoding data. Based on the request/response model, it allows a client to use a distant service whatever technology is used on the client and server. To find out more, you can consult the XML-RPC website (www.xmlrpc.com) which gathers documents, tools and a list of available services.

Using XML-RPC

To use XML-RPC with Rebol, it is necessary to download the latest archive from Andreas Bolka's site. This allows you to write XML-RPC services based on cgi technology as well as clients in a minimum number of lines of code. This library uses Gavin F. McKenzie's XML parser which you must include in your projects.

To write a client, you will need to use five files which are `xml-object.r`, `xml-parser.r`, `xmlrpc-client.r`, `xmlrpc-marshaler.r` and `xml-writer.r`. For Rebol, a service corresponds to an object which is inherited from `xmlrpc-client`.

Using the `set-server` method, you must provide the URL of the server providing the service. Any proxy used can be indicated using the `set-proxy` method. Once the configuration is complete, all there is left to do is request execution of the remote service using the `exec` method. It takes a block containing the name of the remote method and the list of parameters required by it. In the following example, you use the covers.wiw.org service which is a database of song covered by other artists:

```
remote: make xmlrpc-client [ ]
remote/set-proxy tcp://mon-proxy.domaine.org:8080
remote/set-server http://covers.wiw.org:80/RPC.php
print mold remote/exec [ covers.Covered “peter gabriel” ]
```

When calling the method `covers.Covered`, the code searches for the name of artists who have covered songs performed by Peter Gabriel. You are returned a standard Rebol block which you can manipulate with standard Rebol words.

Simple and effective, RebXR is a perfect example of the how Rebol and XML complement each other.

The Rebol/View plugin

The availability of a browser plugin version was a long-awaited event in the Rebol programming community. Its existence creates many possibilities for developing X-internet applications. This vision of an active network in which lightweight applications, dialects and data are readily exchanged between machines can finally become a widely used reality.

Small yet powerful

Such a product obviously bears comparison with Java and its applets.

However this plugin has many advantages that can make it a future star of the internet. The principle difference between the Rebol plugin and Java is the massive difference in size since the plugin is only 600 kb.

This lightweight software can be downloaded and installed in a few seconds on a client machine. Even if the user only has a slow dial-up connection, deploying the plugin does not really pose any difficulty. Don't think for a minute that the small size of the plugin is due to limited functionality! The plugin has all the functionality of the latest version of Rebol/View and thus provides all its possibilities; network protocols, user interfaces, graphics, animation and sound. The programmer has full access to the richness and versatility of Rebol. It is no longer just a matter of adding some cute, small animation to a web page but creating full applications within HTML documents.

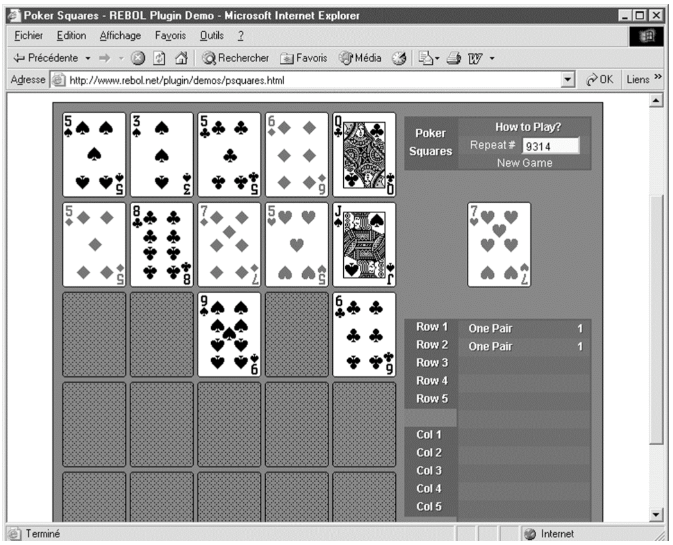


Figure 4-12. *The plugin allows the creation of full applications.*

These reblets have the advantage of being extremely compact and can be downloaded in a few seconds. Their operation is made safe by a virtual machine which uses far less resources than that of Java. Contrary to applets, loading speed and performance should no longer be a barrier.

Inserting the plugin in an HTML page

To test the Rebol plugin, you need a machine running Microsoft Windows. The supported browsers are Internet Explorer, Firefox, Mozilla and Opera. For the moment, the plugin is available for these configurations which are considered a priority by Rebol Technologies. The choice is justified by the fact the Microsoft Windows is currently the most used operating system and, initially, the objective is to provide the product to the maximum number of people. Unix and Mac users and those in love with free software should be reassured that ports of the plugin to Linux and Mac OS X are planned.

Using the plugin simply requires inserting an `<object>` tag within an HTML page. This tag refers to a component downloaded by the browser which contains the Rebol/View 1.3 interpreter. With the first remote use of the plugin, the user is invited to accept its installation with the help of a single click. The browser is then ready to carry out the Rebol script indicated in the HTML page.

```
<object id="myplugin" width="200" height="100"
classid="CLSID:9DDFB297-9ED8-421d-B2AC-372A0F36E6C5"
codebase="http://www.rebol.com/plugin/rebolb5.cab#Version=0,5,0,0">
<param name="LaunchURL" value="http://localhost/script1.r">
</object>
```

The `<object>` tag uses various attributes to provide control over a number of aspects of the plugin. The `id` attributes contains the name the programmer has given to the plug-in in the HTML page. The `width` and `height` attributes represent the width and height of the window to be used by the plugin. The `classid` is the signature of the plugin which, obviously, should not be changed. The `codebase` attributes gives the address from where the plugin can be downloaded if it hasn't already been installed or needs updating. The `LaunchURL` attribute within the `<param>` tag gives the location of the Rebol reblet that is to be executed by the plugin.

In the listing above, the HTML page requests the execution of the program `script1.r`. This is simply a traditional Rebol script using VID (*Visual Interface Dialect*) to describe its graphic interface.

For example, all that is needed to display some text in the plugin window is the following code:

```
REBOL [
 author: "Olivier Auverlot"
]
view layout/size [ title "Reblet !" ] 300x100
```

Configuration Parameters

The plugin provides a number of configuration options that can be accessed via the `<param>` tag. With `BgColor`, you can set the background colour. To do this, you simply provide a hexadecimal value of the RGB components just as you can specific colours in HTML or CSS. Normally, the value used is the same one used for the background of the HTML page that contains the Reblet.

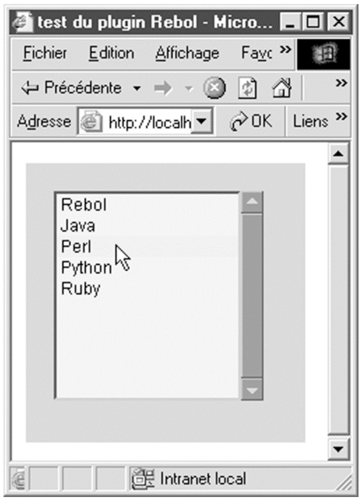
It is even possible to transmit data to a Reblet to configure its operation. For this, you have the `Args` parameter which contains a list of values which is read by the Reblet before it runs. The values contained in this attribute can be dynamically generated using a server-side script or program (CGI, page RHTML or PHP, JSP or Servlet, etc.). To read the contents of the `Args` parameter, the method is the same as that used to read parameters passed to the interpreter from the command line. You just access the `system/options/args` property.

The following example sets the background colour of the plugin and is passed a list of programming languages via the `Args` parameter.

```
<html>
<head><title>Rebol plugin test</title></head>
<body>
<object id="myplugin"
classid="CLSID:9DDFB297-9ED8-421d-B2AC-372A0F36E6C5"
codebase="http://www.rebol.com/plugin/rebolb5.cab#Version=0,5,0,0"
width="200" height="200">
<param name="LaunchURL" value="http://localhost/script2.r">
<param name="BgColor" value="#000000">
<param name="Args" value="Rebol Java Perl Python Ruby">
</object>
</body>
</html>
```

Reading the data from the command line and displaying it in a list only takes a handful of lines of code. After verifying the presence of the data in the `system/options/args` property, the language names are separated from one another using the standard separation character (a space). The different strings are placed in a block and then displayed using the `text-list` style.

```
if not none? system/options/args [
      languages: parse system/options/args ""
]
view layout/size [
      text-list 150x150 data languages
] 300x200
```


test du plugin Rebol - Micro...
Fichier Edition Affichage Favc
Précédente
Adresse http://localh
OK
Liens
Intranet local

Rebol
Java
Perl
Python
Ruby

Figure 4-13. *Running a Reblet in a browser.*

A third and final parameter, called `Version`, is also available. This corresponds to a number of properties available in the Rebol plugin to optimise the transfer of data between the client and the server.

Cache, proxy and compression

In order to make reblets as interactive as possible, Rebol equipped its plugin with an astute caching mechanism to make it possible to optimise the downloading of reblets over the network. When a reblet is used for the first times, its code is stored in a temporary folder on the client computer and within a directory tree dedicated to Rebol.

So under Windows with the Internet Explorer browser, you can find the stored reblets in the `c:\windows\temp\rebol\plugin\ie\0\public\` folder. The scripts are stored in folders named after the server from which they were downloaded. In order to refresh these scripts should they have been updated on the server, the Rebol plugin uses a parameter called Version with the following syntax:

```
<param name="Version" value="3.0.0">
```

If the value of this attribute is higher than the version number of the reblet held in the cache, the reblet on the server is considered to be a new version and will be downloaded. If it is not, the stored reblet will be executed. This easy method has many advantages. By using the `Version` parameter, the developer of a web page containing a reblet can ensure the latest version is used by all users. This cache makes it possible to considerably reduce network traffic by limiting downloads only to reblet updates. Finally, the start-up time of the reblet is much faster as it is stored locally.

The presence of the cache also offers new possibilities to the developers of browser-hosted applications. In effect, the cache can also be thought of as filespace reserved for the Rebol plugin. Files can be created and read in this dedicated tree structure.

This only applies to the dedicated Rebol directories and nowhere else on the user's hard disk. There is no question of reading or writing files outside the dedicated directory structure without Rebol's security manager requesting the user's authorisation. The only accessible directory is the cache.

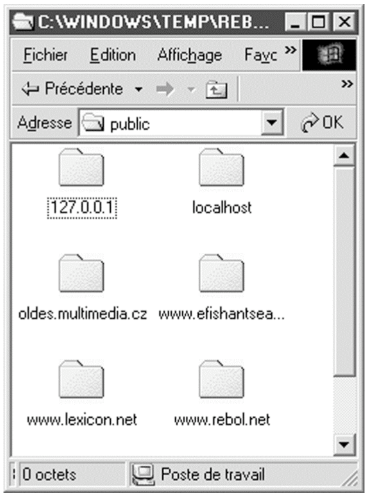


Figure 4-14. *The plugin's local cache*

In this way a reblet can create temporary files, store data on the local disk and even generate HTML or pdf documents. The following example displays a form to capture the user's name and first name. Once the button "Save" is selected, the data are saved on the client in the file called `data.txt`. This file is created in the same folder as the reblet.

```
view layout/size [
across
lab "Name:" name: field "" return
lab "First Name:" firstname: field "" return
btn "Save" [
write %data.txt reduce [
(get-face name) " " (get-face firstname)
]
]
] 400x200
```

To complement the cache, the Rebol plugin provides other tools to considerably reduce network traffic and reduce reblet start-up time.

These are actually all mechanisms available in all versions of Rebol but which take a new dimension under the plugin. For instance, `read-thru` allows optimised reading of documents over a network. Thanks to this, a reblet can easily read the contents of a file at a specified URL. `read-thru` also uses the Rebol plugin cache. After connecting to the server, `read-thru` checks the local cache to see if the requested file is already there. If so, the file is read directly from the cache on the client machine.

For example, a reblet can download images and store them in the local cache. With the next use of the reblet, it will not be necessary to download the images again. It is well worth noting that if you use `read-thru` the data is automatically saved in the cache. On the other hand, if certain information must remain hidden on the server, the programmer can always use `read` and decide what does and what does not need to be saved on the client themselves.

The Rebol plugin also includes the `compress` and `decompress` words for compressing and expanding data. The size of the resources needed to run a reblet (images, sounds, etc.) can be significantly reduced and hence reduce their download time. To compress your files, all you need to use is a Rebol interpreter. The following command compresses the image `img.bmp` and creates a file called img.dat which can then be saved on the web server:

```
write/binary %img.dat compress read/binary %img.bmp
```

You can now write a reblet which downloads it, decompresses it, saves it in the plugin cache and displays it in a window in the browser:

```
img: load to-binary decompress read-thru http://localhost/img.dat
view layout/size [
      origin 0x0
      image 179x250 img
] 179x250
```

As usual, Rebol does so much with the minimum of code.

It is even possible to further reduce the size of the script by using the /`expand` *refinement* of `load-thru` to avoid using the word `decompress`.

Interacting with the Browser

It is possible to interact with the contents of the web page hosting a reblet by accessing the DOM (*Document Object Model)*. This makes it possible to read or modify certain aspects of the environment the reblet is running in. In fact, all of this is made possible by calling a generic JavaScript function whose task is simply to execute JavaScript code provided by the reblet.

This function must be called evaluate and be stored in the page containing the plugin:

```
function evaluate(code) {
      return eval(code);
}
```

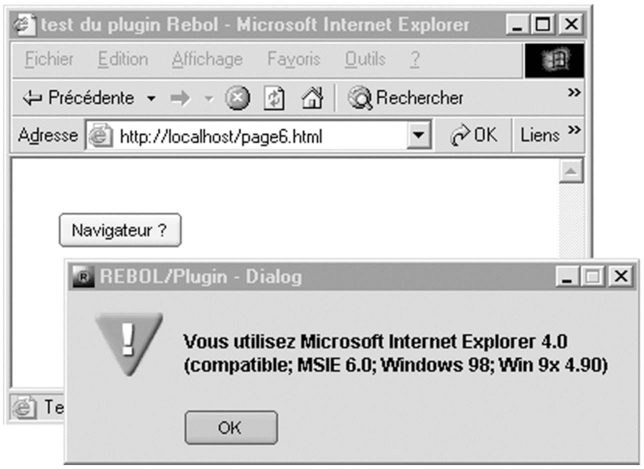
This function performs and returns the result of JavaScript code passed to it as a parameter. In the plugin code, the call of the evaluate() function is made possible using the `do-browser` word whose only argument is the JavaScript code to be passed to the evaluate() function. This word returns the result of the JavaScript code. To better understand, here is a reblet that finds the name and version number of the browsers it is running under. For this, the `appName` and `appVersion` properties of the JavaScript `Navigator` object must be accessed. The HTML page containing the call to the reblet also contains the JavaScript which is stored between the `<head></head>` tags.

```
<html>
<head>
<title>Rebol plugin test</title>
<script language="javascript">
      function evaluate(code) {
            return eval(code);
      }
</script>
</head>
<body>
<object id="myplugin"
classid="CLSID:9DDFB297-9ED8-421d-B2AC-372A0F36E6C5"
codebase="http://www.rebol.com/plugin/rebolb5.cab#Version=0,5,0,0"
width="100" height="50">
<param name="LaunchURL" value="http://localhost/script6.r">
</object>
</body>
</html>
```

The reblet `layout` contains a simple button that when pressed calls the JavaScript `evaluate()` function twice to request the evaluation of the `navigator.appName` and `navigator.appVersion` properties.

Once the information has been collected, it is displayed using a dialog box.

```
view layout/size [
      backdrop 255.255.255
      btn "Browser ?" [
            app-name: do-browser {navigator.appName}
            app-version: do-browser {navigator.appVersion}
            alert join "You are using " [
                  app-name " " app-version
            ]
      ]
] 100x50
```


test du plugin Rebol - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente
Rechercher
Adresse http://localhost/page6.html
OK
Liens
Navigateur ?
REBOL/Plugin - Dialog
OK

Vous utilisez Microsoft Internet Explorer 4.0 (compatible; MSIE 6.0; Windows 98; Win 9x 4.90)

Figure 4-15. *Displaying the Browser details.*

This mode of operation is simple and has the advantage of not imposing any limitations. Anything that can be done in JavaScript can be done with the Rebol plugin. This feature gives the Rebol plugin access to all the resources of the browser.

Through this method, the plugin can read or modify the values of an HTML form's fields, open or close browser windows, consult the navigation history, handle cookies, etc.

By including all the functionality of Rebol/View in a browser plugin, Rebol has certainly made a great coup. Because of its freedom and portability, the plugin resolves the problems of distributing Rebol applications whilst providing the full functionality on client workstations, be they used by professionals or the general public. This light and innovative solution should attract many developers.

Summary

Rebol is truly a language dedicated to network programming. With it, you can access the resources of the Web, define your own data exchange protocols and develop both client/server and web applications. Using reblets, the Internet takes a very different form; that of an organic network composed of nodes exchanging information and applications.

5
Rebol for pros

Rebol has many features to attract professional developers. It has a version dedicated to *e-business* and another to *groupware*. Its versatility, conciseness and portability make it an ideal tool for rapid software development.

Rebol/Command

The Rebol language is currently available in five different versions, each one targeted at a specific use.

Rebol/Core is principally used for network utilities and CGI scripts. Rebol/View provides for the development of graphical client applications. The Rebol/SDK supports the development of commercial applications that can be distributed in executable form. Rebol/IOS is intended for *groupware*. Rebol/Command is intended for developing servers for *e-business* systems.

Concentrated power

Rebol/Command is available for a number of operating systems (Linux, Windows, AIX, etc.). The current version is 2.5.6. It is functionally equivalent across all platforms, just like Rebol/Core and Rebol/View. You have the ability to develop an application without needing to worry about which platform it will run on. Software can be written under Windows and run, unchanged, on Linux.

In a few hundred kilobytes of the Rebol/Command interpreter you find not only all of the words and protocols of Rebol/Core but also an impressive number of extensions which make this product incredibly productive. For web applications intended to be run over the internet or an intranet, there is support for SSL (*Secure Sockets Layer*) for secure data exchange and FastCGI for developing high-performance applications. Rebol/Command also includes access to the principal database management systems available in the market, the ability to call native code libraries, launching external applications via the host system Shell and the main encryption algorithms.

Database access

The first of Rebol/Command's many talents is the ease with which you can communicate with a DBMS. It takes only six lines of code to open a connection, to send a request SQL, to receive the answer and close the connection. Depending on the platform it is running on, Rebol/Command unfortunately doesn't have access to all the same basic databases. Whilst all versions of Rebol/Command natively support Oracle and MySQL, only the Windows version has access to the ODBC protocol. So if you use a different database (Informix, PostgresSQL, SyBase, IBM DB2, etc.), for the moment you must use a Windows machine (9x, Me, NT, 2000, XP).

Database access is achieved with the help of three Rebol protocols called `oracle`, `mysql` and `odbc`. So all you need to do to communicate with a database is to use a `port`. The first stage consists of opening a connection with parameters transmitted in the form of a URL. The parameters will be different depending on the type of database you use.

For example, in the case of ODBC you will have to supply the data source name (DSN) whereas for MySQL you need to give the database name.

Suppose you want to connect to a MySQL database named "test" hosted on the machine at 172.29.143.1 on your network. You must, of course, supply a user name ("homer") and password ("duff"). The connection is established in the following manner:

```
db: open mysql://homer:duff@172.29.143.1/test
```

You can now establish a port by using the word `first`. After that, you transmit your SQL requests to the database with the word `insert` and get the results back from the database with the Rebol word `copy`. Once these operations have been completed you must close the connection by closing the two communications ports:

```
p: first db
insert p "SELECT * FROM myfile"
print copy p
close p
close db
```

This example returns all the rows in the myfile table in the form of a block of blocks which you can then process with Rebol functions. Rebol/Command also supports more complex database operations such as transactions and accessing table and field definitions.

Shell access

The shell is your system's command interpreter. In Linux, it is normally bash (Bourne Again SHell). It allows you to communicate with your machine without the need for a graphical user interface.

A shell provides internal commands (`echo`, `set`, etc.) and lets you launch external applications. Rebol/Command makes it possible for developers to interact with the shell through the `call` word. Suppose that you are running Rebol on Linux, you can obtain the contents of the root directory by using the command `call "ls /"`.

This word has some very interesting *refinements*. With `/error`, you can recover the error code returned by the shell. You can also redirect standard input and output streams with `/input` and `/output`. With `/wait`, it is possible to synchronise the execution of different processes: a task will only be launched once another has finished.

For those of you who haven't bought a copy of Rebol/Command yet, this feature has been included in Rebol/Core and Rebol/View since the release of versions 2.6 and 1.3 respectively.

Using dynamic libraries

A dynamic library is a collection of native functions (and possibly data) which can be shared between applications on the same machine. Library functions are marked either public or private and are correspondingly accessible or hidden from applications. The goal is to avoid having to include the same code directly in every application which uses it. This not only reduces their memory usage but also makes updates easier: it is sufficient to update a library so that all software using it is updated simultaneously.

On Windows, dynamic libraries have the DLL file type. As for Tux fans, they find them with the equally famous SO file type.

Rebol/Command can use native code functions in these dynamic libraries which are usually developed in C, C++, Pascal or even Visual Basic. The steps are rather simple: you must first include the library by using the `library` *refinement* of the word `load`, and then declare the different functions you want to use with the `routine!` datatype. The functions are then regarded as belonging to the standard Rebol dictionary.

Thanks to dynamic libraries, you can re-use native code functions developed for another project, improve the performance of your application by coding critical sections in native code, and especially extending Rebol's capabilities by integrating libraries such as those authorising access to a LDAP directory or a DBMS not directly supported by Command.

Even better news for those of you who haven't bought a copy of Rebol/Command yet, library access has been included in Rebol/View with the release of version 2.7.6.

Data encryption

Data encryption is one of the most interesting aspects of Rebol/Command. Thanks to it, you can encrypt and decrypt information and hence safely transmit it over a network or the internet. The interpreter has a special port named `crypt` whose function is to transform the data according to the selected algorithm. You can use algorithms based on a traditional symmetrical key and also the RSA (Rivest, Shamir and Adelman) algorithm using a pair of keys (private and public). It is also possible to apply a digital signature (DSA) in order to authenticate data. The DH algorithm (Diffie Hellman) is used in the process of authorising connection of outside devices to a network. For all these different cases, Rebol/Command provides the tools needed to generate the keys used. Using these features, you can transmit sensitive information over a network with optimal safety.

In spite of its small size, Rebol/Command is an extremely powerful tool. It has all the functionality needed to quickly write *e-business* applications in a minimum of code. If by chance a required function is missing, you have the alternative of using external applications via the shell access, or better still to include dynamic native code libraries in your project.

Rebol, CGI scripts and MySQL

How to access a database when you don't have Rebol/Command? Indeed, the free versions (Core and View) cannot send SQL requests to a relational database. Happily, the situation is not completely desperate.

With Rebol being a network programming language based on TCP/IP, the solution lies in writing a dedicated communications protocol. Nenad Rakocevic has created and made available an excellent protocol, MySQL.

Introducing the MySQL protocol

Nenad Rakocevic's MySQL protocol works on all versions of Rebol. Used in conjunction with Rebol/Core, it lets you write compact CGI scripts that can interact with the well known MySQL DBMS.

With Rebol/View, you can write office automation applications accessing MySQL or Reblets (distributed network applications). Nenad's protocol is free and can even be integrated into commercial applications. The protocol is totally independent from the executing platform and so can be used on many operating systems (Linux, Mac OS X, Windows, etc.).

If you don't want to buy Rebol/Command, using Rebol/Core and MySQL provides an excellent alternative for developing *e-business* applications.

Downloading and installing the protocol

Nenad Rakocevic's MySQL protocol is available for downloading from http://softinnov.org/rebol/mysql.shtml. (You can also find a link to his similar PostgresQL protocol on this page.)

The protocol is contained in a ZIP archive which contains a comprehensive developer's guide.

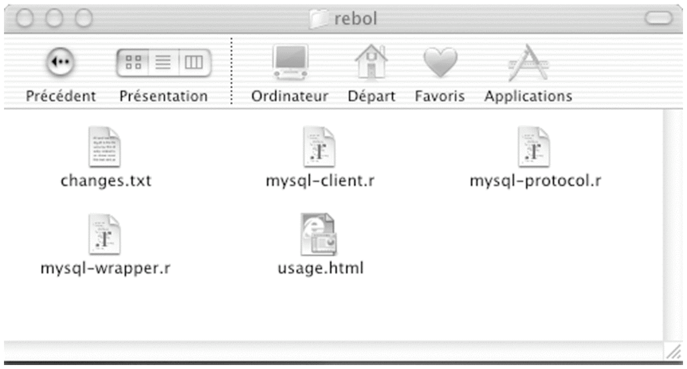


Figure 5-1. *The contents of the MySQL protocol folder.*

Among the new files on your hard drive, the two most important are `mysql-usage.html` and `mysql-protocol.r`.

The first is the complete documentation of the protocol, the second the code to include in your application to be able to communicate with MySQL. To include it in your program, you simply run the script containing the protocol by using the word `do` followed by path to the file containing the protocol. Once the protocol is loaded, it displayed the message "MySQL protocol loaded" in the Rebol console. You can now check that the new protocol is available by using the command `print mold first system/schemes`.

Using the protocol

Now all that is left is to test the protocol. For this, you will create a small MySQL database called `musicdb` which contains only one table `discs`. Its purpose is to store details of your record collection. It consists of five fields:

```
CREATE TABLE discs (
      num SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
      artist CHAR(60),
      title CHAR(60),
      year SMALLINT,
      PRIMARY KEY (num)
)
```

The database will reside on a Linux server with the IP address of 172.29.143.1. With the help of a Rebol script, you are going to load this table with data. The data representing your disc collection are contained in a block called, `data`.

Each sub-block within the `data` block corresponds to a disc and holds the name of the artist, the title and the year the disc was published.

The connection to the database is made by using the `open` instruction followed by the name of the protocol and a URL in which you specify the user name and password (`olivier:homer`), the IP address (or name) of the server hosting the MySQL DBMS and the name of the database. This instruction then returns a port into which you can insert SQL instructions and retrieve data.

To load the `discs` table, all you need to do is to cycle through the `data` block with a `foreach` loop and then use the SQL instruction `insert into`.

The protocol provides a great convenience for programmers by allowing the substitution of "?" for the values of variables passed as parameters (the first question mark is the first variable and so forth). This cuts down on the construction of a complex string when using application parameters. At the end of the process, don't forget to close the port with the `close` instruction to release it.

```
#!/Applications/core/rebol -qs
REBOL [ ]
do %/Library/rebol/mysql-protocol.r

data: [
      [ "Leonard Cohen" "Leonard Cohen in concert" 1994 ]
      [ "Sarah McLachlan" "Remixed" 2001 ]
      [ "Jean-Louis Murat" "Muragostang" 2000 ]
      [ "Leonard Cohen" "Ten new songs" 2001 ]
      [ "Peter Gabriel" "Us" 1992 ]
      [ "Huong Thanh" "Moon and Wind" 1999 ]
]

db: open mysql://olivier:homer@172.29.143.1/musicdb
foreach elem data [
      insert db [ "insert into discs values (0,?,?,?)"
      elem/1 elem/2 elem/3 ]
 ]
close db
```

Integration in a CGI script

To query your disk catalogue, you are going to write a short CGI script called `music.cgi`. The user must select an artist from a dropdown menu to get a list of the artist's albums.

The only difficulty with this script is to properly handle the information received from the form. The script is recursive and any parameter will be sent to it on its first execution. To determine the receipt of an artist's name, you must use error handling: the code tries to read the information supplied and assigns the result to the variable, `artist-name`.

If the script is unable to find the `artist` field from the input data, it considers that no parameter was transmitted and initialises `artist` with the value `none`.

```
REBOL [ ]
do %/Library/rebol/mysql-protocol.r
print "Content-type: text/html^/"
print {
      <HTML>
      <HEAD><TITLE>Music</TITLE></HEAD>
      <BODY>
      <H1>Disk catalogue</H1>
}
if error? try [
      query: make object! decode-cgi system/options/cgi/query-string
      artist-name: query/artist
] [artist-name: none ]
```

In any case, you should see a selectable drop-down list containing the names of the artists in the database.

To avoid duplicates, the SQL request uses the `distinct` option and names obtained are sorted in alphabetic order (`order by`).

Because the application doesn't know how many artists are in the database, and it could be a very large number, the data are read one by one from the port by using the word `first`. They are dynamically added to the HTML form.

```
print {
<FORM name='catalogue' action='music.cgi' method='GET'>
<SELECT name='artist' onChange="catalogue.submit();">
<OPTION>Choose an artist
}
db: open mysql://olivier:homer@172.29.143.1/musicdb
insert db [
      "select distinct artist from discs order by artist"
]
while [ not empty? (name: first db) ] [
      print join "<OPTION>" name
]
print "</SELECT></FORM>"
```

The last stage consists of querying the database if an artist has been selected.

There is no need to reopen a connection as you are always connected to the MySQL database. The result of the `select` request is assigned to the variable `discs` which is a block of blocks containing a block for each line in the response.

An HTML table is built and the data is displayed in it by using a `foreach` loop. All that is left to do is close the database connection and to complete the construction of the HTML page.

```
if not none? artist-name [
      insert db [
            "select title,year from discs where artist=?"
            artist-name
      ]
      discs: copy db
      print [ "<BR><B>" nom-artiste "</B><BR><TABLE>" ]
      foreach d discs [
            print [ "<TR><TD>" d/1 "</TD><TD>" d/2 "</TD></TR>" ]
      ]
      print "</TABLE>"
]
close db
print "</BODY></HTML>"
```

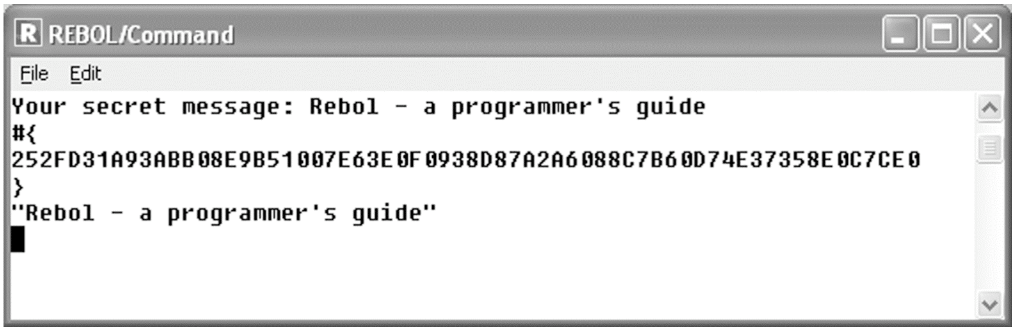
Thanks to the tremendous work of Nenad Rakocevic, your CGI scripts, written with Rebol/Core are now capable of interacting with a MySQL database. As you may have noticed, performing SQL queries and reading the data are easy.

Data encryption

The primary goal of Rebol is to facilitate the transport of data between heterogeneous computer systems. Moving information over networks requires taking into account a crucial factor: security. A communications language must have the necessary tools to ensure confidentiality and integrity of data. To this end, there are many of the major encryption algorithms available in Rebol/View/Pro, Rebol/SDK, Rebol/Command and Rebol/IOS.

Symmetric key encryption

This type of encryption is based on a single key. The recipient decrypts the message with the same key that the sender used to encrypt it. This method is very simple but it has a significant disadvantage: transporting the key. In fact, for the sender and recipient to have the same key, at one time or another, the key must have travelled between the two users. Additionally, for all users who have the same key, this technique does not authenticate the documents or even verify the integrity of data. Nevertheless, this method is extremely fast and if you know the limitations, it can be used effectively in certain circumstances.



```
Your secret message: Rebol - a programmer's guide
#{
252FD31A93ABB08E9B51007E63E0F0938D87A2A6088C7B60D74E37358E0C7CE0
}
"Rebol - a programmer's guide"
```

Figure 5-2. *Symmetric key encryption is simple and fast.*

Rebol includes a protocol named `crypt` for data encryption.

You simply define encryption parameters and enter data into port to encrypt and decrypt information. The following example encrypts a message input by the user.

First a 128-bit key (16 bytes) is generated from a string.

```
the-key: checksum/secure "RebollobeRReboll"
```

Then the user is asked to input a message:

```
msg: ask "Your secret message: "
```

Next a port is defined using the `'encrypt` function of the `crypt` protocol. The algorithm can be either of the `'blowfish`, or `'rijndael` types.

The `padding` property ensures compatibility with other encryption applications

```
cp: make port! [
      scheme: 'crypt
      algorithm: 'blowfish
      direction: 'encrypt
      strength: 128
      key: the-key
      padding: true
]
```

The message is encrypted when it is inserted into the `cp` port. You simply use the word copy to retrieve the encryption results.

```
open cp
insert cp msg
update cp
encrypted-msg: copy cp
close cp
print mold encrypted-msg
```

To decrypt the message, you must use the same method (and especially the same key!) by changing the action of the protocol to `'decrypt`.

```
cp: make port! [
      scheme: 'crypt
      algorithm: 'blowfish
      direction: 'decrypt
      strength: 128
      key: the-key
      padding: true
]

open cp
insert cp encrypted-msg
update cp
print mold to-string copy cp
close cp
```

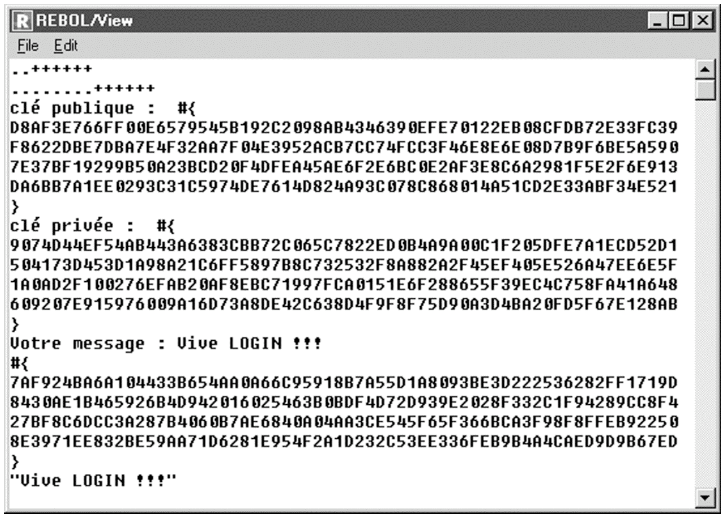
RSA encryption

RSA (Rivest, Shamir and Adelman) encryption is based on the use of two keys linked to one another by a mathematical relationship:

- the private key is never revealed or transmitted,
- the public key is sent to all recipients.

This algorithm not only protects the contents of a message but also authenticates its author. To encrypt a message, the sender encrypts the message with the recipient's public key (which is known to all). The latter decrypts the message with their private key. The authentication of the message is made possible by the confidentiality of the private key. If the sender uses his private key to encrypt the message, the recipient has no choice but to use the sender’s public key to access information. If the message is successfully decrypted, it proves the message was sent by the named sender. This method also avoids repudiation: issuers cannot claim that they didn’t send a message.

One aspect of RSA encryption is that the length of the message cannot exceed the length of the keys. Also RSA is slower than symmetric algorithms. For these reasons, the main practical use of RSA encryption is to exchange temporary keys for use with symmetric algorithms and the “signing” of documents.



```
..++++++
........++++++
clé publique :  #{
D8AF3E766FF00E6579545B192C2098AB4346390EFE70122EB08CFDB72E33FC39
F8622DBE7DBA7E4F32AA7F04E3952ACB7CC74FCC3F46E8E6E08D7B9F6BE5A590
7E37BF19299B50A23BCD20F4DFEA45AE6F2E6BC0E2AF3E8C6A2981F5E2F6E913
DA6BB7A1EE0293C31C5974DE7614D824A93C078C868014A51CD2E33ABF34E521
}
clé privée :  #{
9074D44EF54AB443A6383CBB72C065C7822ED0B4A9A00C1F205DFE7A1ECD52D1
504173D453D1A98A21C6FF5897B8C732532F8A882A2F45EF405E526A47EE6E5F
1A0AD2F100276EFAB20AF8EBC71997FCA0151E6F288655F39EC4C758FA41A648
609207E915976009A16D73A8DE42C638D4F9F8F75D90A3D4BA20FD5F67E128AB
}
Votre message : Vive LOGIN !!!
#{
7AF924BA6A104433B654AA0A66C95918B7A55D1A8093BE3D222536282FF1719D
8430AE1B465926B4D942016025463B0BDF4D72D939E2028F332C1F94289CC8F4
27BF8C6DCC3A287B4060B7AE6840A04AA3CE545F65F366BCA3F98F8FFEB92250
8E3971EE832BE59AA71D6281E954F2A1D232C53EE336FEB9B4A4CAED9D9B67ED
}
"Vive LOGIN !!!"
```

Figure 5-3. *RSA uses both a private and a public key.*

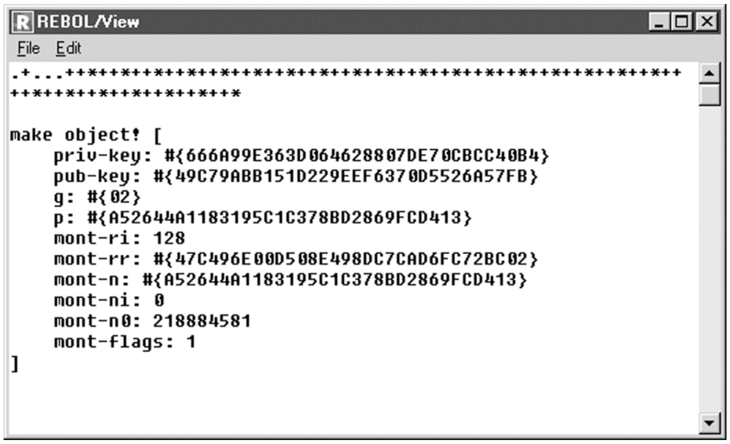
For storing the two keys, Rebol uses an object created by the word `rsa-make-key`. Then the keys can be generated (`rsa-generate-key`) by specifying the length of the keys and a prime number to be used by the algorithm. The properties n and d of the object containing the keys are the public key and private key respectively.

The word `rsa-encrypt` can then use these two keys to encrypt and decrypt data.

```
keys: rsa-make-key
rsa-generate-key keys 1024 3
print [ "public key: " (mold keys/n) ]
print [ "private key: " (mold keys/d) ]

msg: ask "Your message : "

msg-code: rsa-encrypt/private keys (to-binary msg)
print mold msg-code
msg-decode: rsa-encrypt/decrypt keys msg-code
print mold to-string msg-decode
```

Diffie Hellman

The D-H algorithm can establish secure connections over open networks. The principle is also based on generating two keys. Each side has a private key that is never broadcast or moved. They also have public keys whose purpose is to be used between the machines between which secure communications must be established.



```
.+...+++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++

make object! [
    priv-key: #{666A99E363D064628807DE70CBCC40B4}
    pub-key: #{49C79ABB151D229EEF6370D5526A57FB}
    g: #{02}
    p: #{A52644A1183195C1C378BD2869FCD413}
    mont-ri: 128
    mont-rr: #{47C496E00D508E498DC7CAD6FC72BC02}
    mont-n: #{A52644A1183195C1C378BD2869FCD413}
    mont-ni: 0
    mont-n0: 218884581
    mont-flags: 1
]
```

Figure 5-4. *Generation of two keys with D-H*

Following the exchange of public keys, the two correspondents generate a session key to be used for symmetric encryption between the two parties.

The use of this encryption method is dictated simply because of its speed: symmetric encryption allows both parties to encrypt and decrypt data in real time.

Using the D-H algorithm is very simple with Rebol. Generating the two keys for a correspondent is done using the word `dh-make-key` with the `/generate` refinement. The arguments are the key length and a number to seed the key generation. You then need to generate a key pair with the word `dh-generate-key`.

```
keys: dh-make-key/generate 128 2
dh-generate-key keys
```

The final step is to generate a session key from your private key and the public key received from your correspondent

```
; the variable key-other-server contains
; the public key of the correspondent
Session-key: dh-compute-key keys key-other-server
```

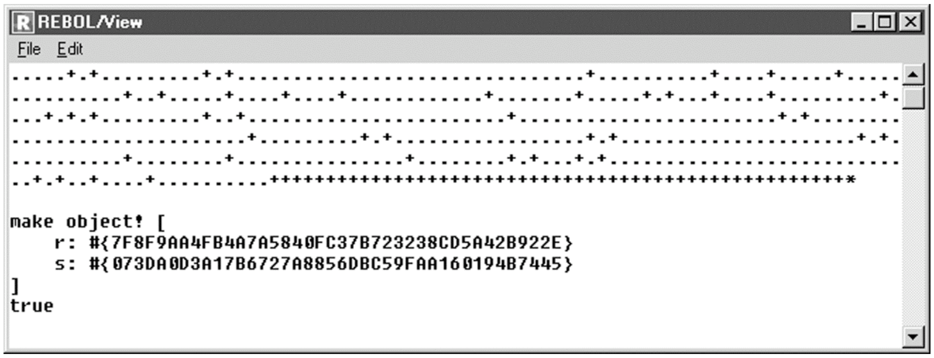
From now, simply use the symmetric encryption method from the beginning of the chapter. The two correspondents will have independently generated the identical session key and can now safely exchange information.

Signing your documents with DSA

DSA (*Digital Signature Algorithm*) allows a document to have a digital signature. It also relies on the use of private and public keys. It is very simple to use as the sender uses their private key to attach their signature to the document. To authenticate the document, the recipient uses the public key of the sender. If there is a match, the recipient can be certain of the author. Don't get confused about the purpose of DSA: it is not here to protect the contents of a message, only to provide a service to authenticate a document's author. For this reason, DSA is not applied to all data but only a portion of it. Anyway, the algorithm cannot be used to encrypt text that is the same length or longer than the key. Because of this, it is generally used on the checksum of a message. The keys are generated with the words, `dsa-make-key` and `dsa-generate-key`.

A signature is created with `dsa-make-signature` and verified by `dsa-verify-signature` which returns a simple boolean value.

```
data: checksum/secure "message to which the signature will be applied"

my-key: dsa-make-key/generate 1024
dsa-generate-key my-key
signature: dsa-make-signature/sign my-key data
print mold signature
print dsa-verify-signature key data signature
```



```
.....+.+.........+.+...............................+..........+....+.....+.....
..........+..+.....+....+....+............+.......+.....+.+...+....+.........+.
...+.+.+.........+..+.......................+.......................+.+........
.....................+.........+.+.................+.+.....................+.+.
..........+........+...............+........+.+...+.+..........................
..+.+..+....+..........+++++++++++++++++++++++++++++++++++++++++++++++++++*

make object! [
    r: #{7F8F9AA4FB4A7A5840FC37B723238CD5A42B922E}
    s: #{073DA0D3A17B6727A8856DBC59FAA160194B7445}
]
true
```

Figure 5-5. *Keys and DSA signature verification.*

As you can see, at the end of this rapid overview of the encryption technologies available in Rebol, it has the best available tools to ensure the safety of your data as it travels over networks.

Rebol/IOS

Rebol/IOS (*Internet Operating System*) is the Rebol application server. Echoing the features of the other versions of the language (Core, View and Command), it allows the creation of a community of users who can easily share documents and applications. Rebol/IOS is an extremely deep product but still incredibly lightweight. It is the sublimation of Rebol concepts: the materialisation of a vibrant, interactive and dynamic Internet.

IOS: a concept lacking exposure

Rebol/IOS allows a group of people to exchange, manipulate and share information.

Using the HTTP transport protocol, it can be used both on intranets and the Internet. Users connect to the server via a dedicated client called Link, which gives a genuine working environment that can be described as a virtual office. Identified by their user account, they can not only send data to and retrieve data from the server but also use the applications it hosts.

The main feature of IOS, apart from the fact that you can totally avoid a cumbersome browser, is that it inherently employs the concept of synchronisation. Whenever a client connects, its environment is updated to reflect all changes made on the server. So if a Rebol application, which is referred to as a reblet, or even the Rebol/Link client itself changes, the client automatically retrieves and installs the new versions.

This mirroring mechanism and the fact that reblets are executed under Rebol/Link on the client, gives the user the choice of working either *on-line* or *off-line*. In the latter case, the client will synchronise any changes made by the user when it next connects to the server. A mobile user can connect to their IOS server from time-to-time and update their data. During the day, they can work in their virtual office using Rebol applications. At night, the user simply connects to the IOS server which automatically uploads their data and updates their working environment if necessary. For the programmer as well as the user, all these operations are made totally transparent and, above all, are carried out safely.

An ultra-secure system

The people from Rebol have really focussed on the security of information transferred between Link clients and the application server. It isn’t necessary to add anything to the product: IOS really knows how to manage its security.

The first thing to know is that a Link client is dedicated to which server it can connect. Both client and server include a certificate which blocks unauthorised attempts to connect. A Link client of company A can never connect to company B’s IOS server (unless of course, both parties want such connections).

Once a connection is established, a Link client and its server share a session key whose exchange is based on the RSA (Rivest, Shamir and Adelman) model. This means that all exchanges between the machines will be automatically encrypted. A person may try snooping on your network, but they will learn nothing.

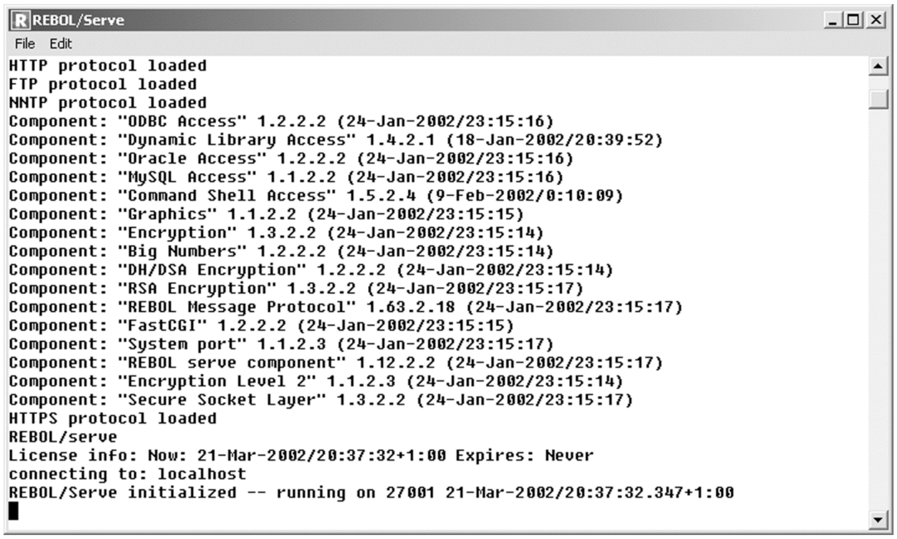
If you push the standard configuration a little further, you can filter IP addresses and TCP ports from which machines are allowed to connect to the server. You can even encrypt data that is stored on the client and server. Rebol/IOS also uses various methods to ensure data integrity such as SHA1 checksums.

On the server, a user has certain rights determined by their username. User identification is based on a classical login with password. Each user can have different rights and they can customise their work environment. You can hide documents and applications from certain users: they will never know the hidden documents or applications existed and have no way of finding out. Moreover, all user and server actions are recorded in a log file that lets you monitor IOS activity and helps you resolve possible problems.

The Rebol/Serve server

The heart of IOS is the server called REBOL/Serve. This product can be installed in five minutes and is available for Windows and Linux.

The ZIP archive of the Windows version contains five directories. The server itself is in the `rebol-serve` directory and is called, in case you couldn't guess, `rebol-serve.exe`. It is a simple executable of around 680K. To activate it, simply type the command `rebol-serve.exe -wqs`. Of course, you can create a batch file to launch it or, on Unix systems, use the command files provided in the `scripts` directory. Simply enter the shell commands `server start` or `server stop` to start or stop the IOS server.



```
HTTP protocol loaded
FTP protocol loaded
NNTP protocol loaded
Component: "ODBC Access" 1.2.2.2 (24-Jan-2002/23:15:16)
Component: "Dynamic Library Access" 1.4.2.1 (18-Jan-2002/20:39:52)
Component: "Oracle Access" 1.2.2.2 (24-Jan-2002/23:15:16)
Component: "MySQL Access" 1.1.2.2 (24-Jan-2002/23:15:16)
Component: "Command Shell Access" 1.5.2.4 (9-Feb-2002/0:10:09)
Component: "Graphics" 1.1.2.2 (24-Jan-2002/23:15:15)
Component: "Encryption" 1.3.2.2 (24-Jan-2002/23:15:14)
Component: "Big Numbers" 1.2.2.2 (24-Jan-2002/23:15:14)
Component: "DH/DSA Encryption" 1.2.2.2 (24-Jan-2002/23:15:14)
Component: "RSA Encryption" 1.3.2.2 (24-Jan-2002/23:15:17)
Component: "REBOL Message Protocol" 1.63.2.18 (24-Jan-2002/23:15:17)
Component: "FastCGI" 1.2.2.2 (24-Jan-2002/23:15:15)
Component: "System port" 1.1.2.3 (24-Jan-2002/23:15:17)
Component: "REBOL serve component" 1.12.2.2 (24-Jan-2002/23:15:17)
Component: "Encryption Level 2" 1.1.2.3 (24-Jan-2002/23:15:14)
Component: "Secure Socket Layer" 1.3.2.2 (24-Jan-2002/23:15:17)
HTTPS protocol loaded
REBOL/serve
License info: Now: 21-Mar-2002/20:37:32+1:00 Expires: Never
connecting to: localhost
REBOL/Serve initialized -- running on 27001 21-Mar-2002/20:37:32.347+1:00
```

Figure 5-6. *The server in action.*

Once the server is running, you must install the proxy. This small application of around 35K is in the `proxy` directory in both source and compiled versions. The different files are for GNU Make and Microsoft Developer Studio, so you can build this program for another platform.

What is the purpose of this proxy? In fact, for IOS to run over the Internet, you need a Web server. It can be almost any Web server; the only condition is that it supports persistent HTTP connections. Rest assured, the two market leaders (Apache and Microsoft IIS) do this very well and are perfectly suited to IOS. The proxy programs goal is to act as an intermediary between Link clients connecting via HTTP to the webserver and IOS which uses a specific TCP port. The proxy is a simple, tiny CGI script, you must place it in the virtual cgi-bin directory of your Web server.

It is very interesting to note the HTTP server and, by extension, the CGI script, do not need to run on the same machine as the IOS server. You can put together a Linux/Apache server to receive connections from clients and a Windows 2000 server hosting Rebol/IOS.

The architecture is very flexible and allows you to use the best products to meet your needs.

The Rebol/Link client

Once the server is working, it will only connect with a Rebol/Link client. It is a lightweight application of about 500K in its Windows version. The configuration needed for it to run well is only a 200 MHz microprocessor, 64 Mbytes of RAM and around 4Mbytes of disk space. Not exactly bloatware! (It has been reported that Rebol/Link runs happily under Wine on Linux). Even with a simple modem, Rebol/Link can be downloaded in a few seconds.

The Rebol/Link client contains its own installation procedure. The latter allows the user to enter their account name and password and information about the network to which it belongs (mail server and proxy).

Once launched, Rebol/Link takes the form of a desktop divided into three zones:

- the top of the screen contains a button bar which allows you to access commonly-used applications,
- the left columns lists the folders for different types of activity,
- the main area of the screen is used to display icons of applications, folders and files.

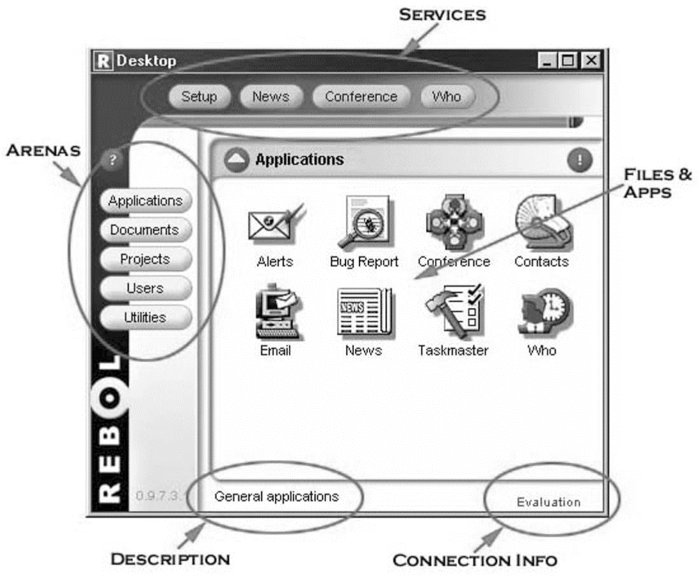


Figure 5-7. *Link client organisation.*

This desktop can be heavily personalised and the graphics adapted to the requirements of a company or organisation. You can change its colours or add a logo. For the administrator, this work consists of creating a *skin* which will be automatically downloaded to clients when they next synchronise.

Using Rebol/Link is very intuitive because it is totally mouse-driven. The user has only to click on the various icons representing the features available. The right mouse button lets you change properties of a document and is also used to publish information on the IOS server. These documents can be files, possibly grouped in folders, hypertext links to HTML pages or applications written in Rebol which run on the client. If you use office documents (Word, Excel, Visio, PDF, etc.), Rebol/Link will use the file associations (.doc etc.) and launch the appropriate application to open the file (on condition that it is available on the user's machine).

Apart from sharing documents within a work group, the primary role of Rebol/Link is obviously to run applications written in Rebol, these are called Reblets.

Reblets

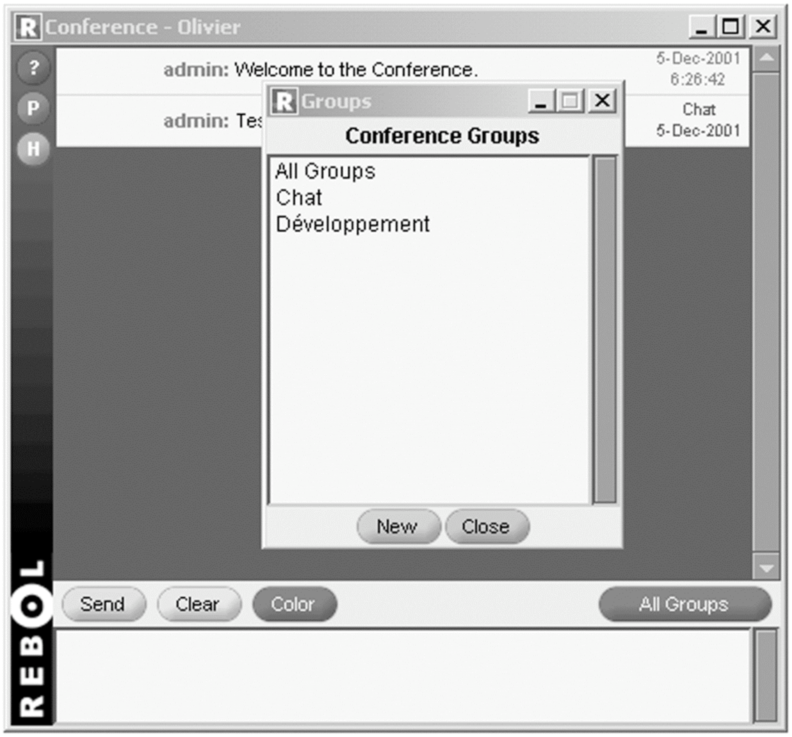
These ultra-light applications are the heart of the Rebol strategy known as the X-Internet. This is not simply the distribution of HTML pages but that of genuine software. Each client Reblet becomes an active part of the network and can exchange information with other members of the community. The applications no longer function solely on the server but are built using a distributed architecture. There are actually three different methods for the division of roles between client and server:

- A reblet may be run exclusively by the client,
- A reblet may be run on both the client and the server: in this case, the reblet uses what is called a `post` function which is located on the server in the `applications` directory,
- A reblet may also trigger the execution of external programs on the server and use the subsequent results.

This integration of the client and server in the process of manipulating and presenting information, results in the burden on the network and server being considerably reducing in comparison with today's conventional solutions (scripts CGI, servlet, JSP, PHP, etc.).

To demonstrate the capabilities of its solution and to position Rebol/IOS in the field of collaborative work, Rebol provides a dozen reblets ready for use and personalisation. The source code of each of them is available and you can freely adapt them to your needs.

Many areas are covered by these reblets and they make IOS a real hub in a network. For instance, in place of normally less secure messaging, the Messenger application allows two user to participate in a direct or deferred dialogue (all messages are archived and therefore you keep the history of your conversations). With Conference, you can create themes for discussion and several users can exchange their views simultaneously. For the nostalgic, Rebol/IOS includes a small reblet for sending messages via the classic SMTP protocol.



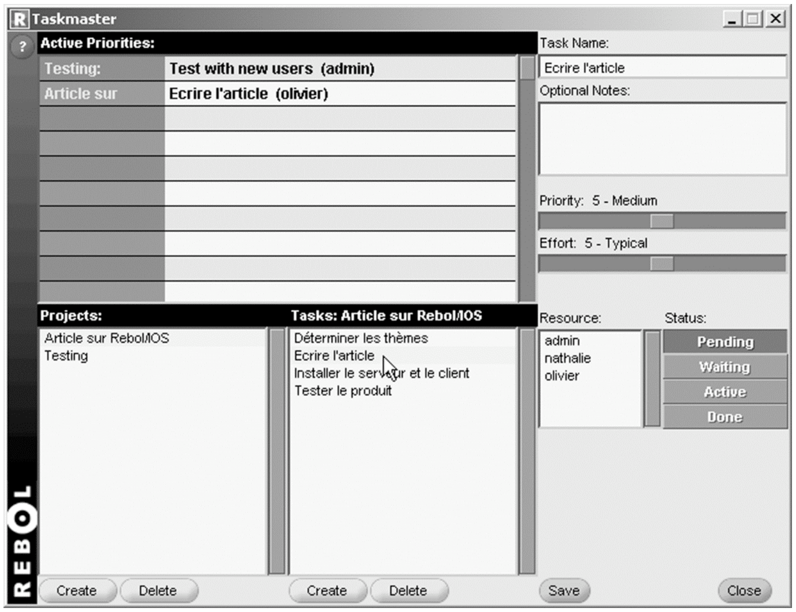
admin: Welcome to the Conference. 5-Dec-2001 6:26:42

admin: Te Chat 5-Dec-2001

Figure 5-8. *The Conference reblet.*

In the light-weight office domain, you have a shared contact manager of which everybody can both view and maintain the contents. A shared calendar (Calendar) and basic project manager (Taskmaster) facilitate the co-ordination of work between different people.

Another very practical application is the Who reblet. This shows who is connected to the server, their availability and location. By simply clicking on one of a few visible buttons, you can inform the members of your community, if you are available, busy, in a meeting or have simply gone home. It is even possible for your colleagues to find out where you are: Who shows whether you are connected from your office or your home.



Active Priorities:

Testing:	Test with new users (admin)
Article sur	Ecrire l'article (olivier)

Task Name: Ecrire l'article

Optional Notes:

Priority: 5 - Medium

Effort: 5 - Typical

Projects:
Article sur Rebol/IOS
Testing

Tasks: Article sur Rebol/IOS
Déterminer les thèmes
Ecrire l'article
Installer le serveur et le client
Tester le produit

Resource:
admin
nathalie
olivier

Status:

Figure 5-9. *Taskmaster is a simple project manager.*

There are numerous reblets and Rebol Technologies announces new ones when they become available; some are included with the server, some are sold separately. You can monitor product sales by using the Sales application. Survey is designed for organising polls which users can participate. You can create curves and histogram from data sources with Plot. Presenter is a program allowing the broadcasting of interactive presentation on a network. These are just a few examples from the list of IOS applications.

Anyway, if you do not find what your are looking for in the list of applications, there is nothing to stop you modifying the applications or developing your own reblets, fully adapted to your needs.

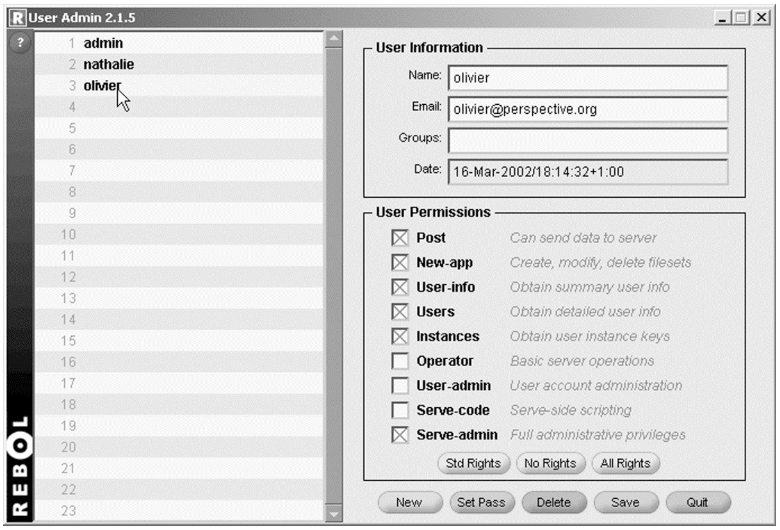
Administration tools

Reblets are so versatile that they are even used to administer the server. You just login to the "admin" account to get access to the Rebol/IOS management functions. An administrator can connect from any computer with a Link client.

Amongst the many tools available, the three main ones are, without a doubt, Alert for sending messages to all users, User-admin for managing user accounts and Reg-edit for updating *filesets* (explained below).

An administrator can broadcast a message to every user who is connected to the server with Alert. On the client desktop, a window will automatically open to display the alert message. This function is very useful to warn users of an important event such as a warning that the machine or a service will be unavailable.

With User-admin, you have an excellent tool for creating and managing user accounts. For each of them, you must enter a name and email address. You can also connect a user to one or more groups and define specific rights for each of the nine base criteria available. The system is very flexible and allows very fine control over each user's rights. You can also import users from another Rebol/IOS system with the help of a simple Rebol script. It is even possible to automatically incorporate elements from LDAP directory services.


User Admin 2.1.5

1 admin
2 nathalie
3 olivier

User Information

Name: olivier
Email: olivier@perspective.org
Groups:
Date: 16-Mar-2002/18:14:32+1:00

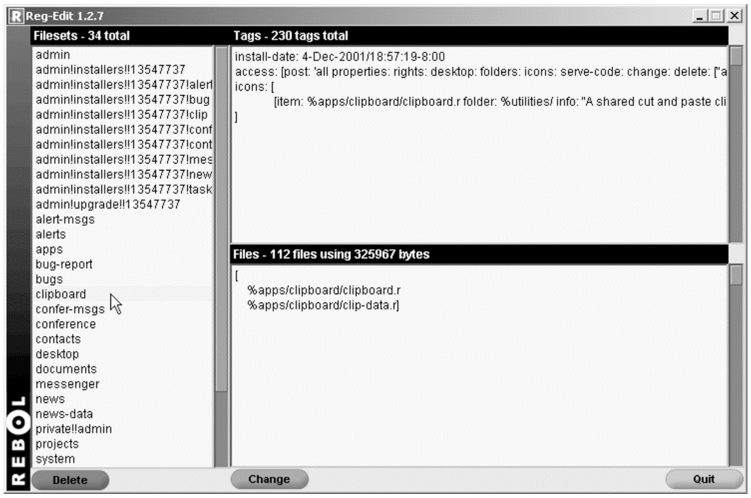
User Permissions

☒	Post	Can send data to server
☒	New-app	Create, modify, delete filesets
☒	User-info	Obtain summary user info
☒	Users	Obtain detailed user info
☒	Instances	Obtain user instance keys
☐	Operator	Basic server operations
☐	User-admin	User account administration
☐	Serve-code	Serve-side scripting
☒	Serve-admin	Full administrative privileges

Figure 5-10. *Managing user accounts.*

Reg-edit is used to administer Rebol/IOS *filesets*, the file system used by the server.

Each file manipulated, whether an application or data, belongs to a *fileset* which defines the files characteristics, access rights, the use to which it can be put and much other information including which icon is to be used on the Link desktop. Publishing a reblet on the server using the POST function, boils down to creating a new fileset on the server.


Reg-Edit 1.2.7

Filesets - 34 total

```
admin
admin!installers!!13547737
admin!installers!!13547737!alert
admin!installers!!13547737!bug
admin!installers!!13547737!clip
admin!installers!!13547737!conf
admin!installers!!13547737!cont
admin!installers!!13547737!mes
admin!installers!!13547737!new
admin!installers!!13547737!task
admin!upgrade!!13547737
alert-msgs
alerts
apps
bug-report
bugs
clipboard
confer-msgs
conference
contacts
desktop
documents
messenger
news
news-data
private!!admin
projects
system
```

Tags - 230 tags total

```
install-date: 4-Dec-2001/18:57:19-8:00
access: [post: 'all properties: rights: desktop: folders: icons: serve-code: change: delete: ["a
icons: [
        [item: %apps/clipboard/clipboard.r folder: %utilities/ info: "A shared cut and paste cli
]
```

Files - 112 files using 325967 bytes

```
[
   %apps/clipboard/clipboard.r
   %apps/clipboard/clip-data.r]
```

Figure 5-11. *Managing the registry database.*

Rebol/IOS is a stunning product with no real equivalent. Simply use it for a few seconds to understand that Carl Sassenrath has created a platform to revolutionise the use of the Internet. Capable of integrating with an existing environment and slowly replacing it, Rebol/IOS can become the heart of an information system. The delivery of solutions built around a Rebol/IOS server, specifically configured in terms of look and function, according to the specific needs of an organisation is undoubtedly a future market for developers.

Managing Rebol projects

The Rebol language is so compact that you can write your programs with a minimum number of lines of code generally in a single script. But for more ambitious projects, the maintenance needs often require the project to be split into multiple files. You must then use the Prebol source manager.

The Prebol preprocessor

The Prebol utility is a preprocessor for Rebol code. It allows the automatic construction of a Rebol script from a set of resources such as files containing Rebol code, data or multimedia files (images, sounds, etc.).

This generation can be static or dynamic according to a set of programmer-defined parameters, the development environment or even the target runtime environment.

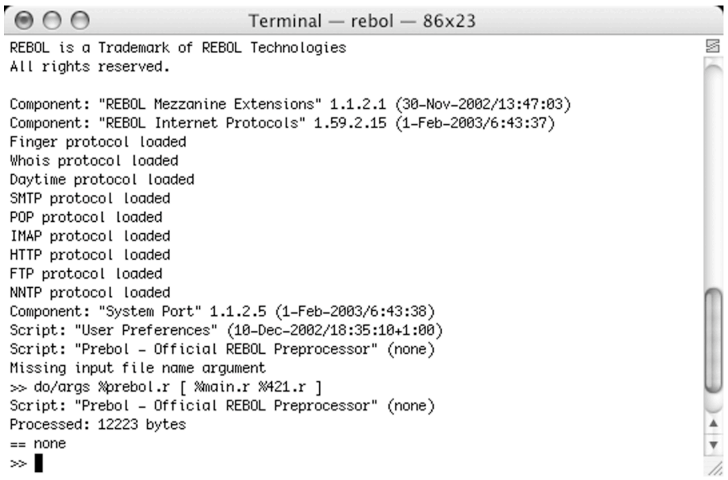
Prebol also allows the optimisation of Rebol code as it minimises the size of the script to be distributed. To do this the utility removes unnecessary characters such as metadata in any files included, comments and some end of line characters. The aim of these operations is to provide a concise file which can be easily distributed via the Internet. Prebol is an essential tool for the Rebol developer.

Installation and use

Before installing Prebol on your machine, you must first get it. You have the choice between paid and free versions. Owners of the commercial Rebol/SDK (Software Development Kit) have this utility in two forms: a binary executable in the `bin` folder and a script in the `source` folder. To invoke it from the command line, you should use the command `prerebol` followed by the name of the script source and the name of the file to be generated.

```
prerebol project.r program.r
```

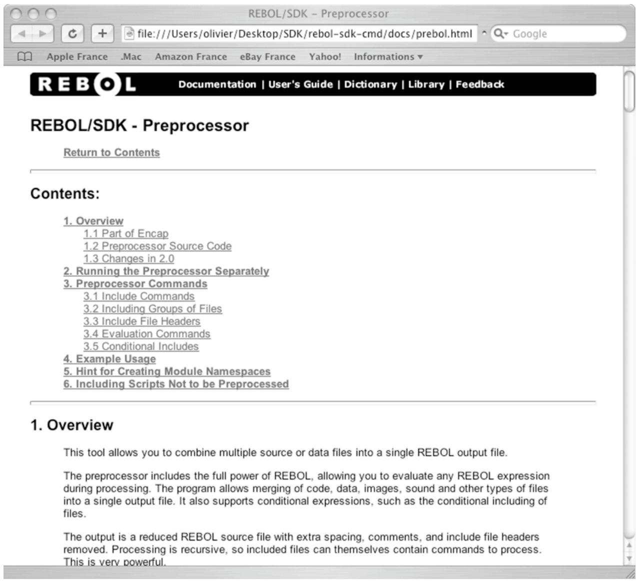
As an alternative to executing it from the shell, you can also use the script `prebol.r` followed by the same arguments. It is also possible to invoke it directly from the Rebol console by using the `do` word with the `/args` *refinement*, the syntax is then `do/args %prebol.r [ %project.r %program.r ]`.



```
Terminal — rebol — 86x23
REBOL is a Trademark of REBOL Technologies
All rights reserved.

Component: "REBOL Mezzanine Extensions" 1.1.2.1 (30-Nov-2002/13:47:03)
Component: "REBOL Internet Protocols" 1.59.2.15 (1-Feb-2003/6:43:37)
Finger protocol loaded
Whois protocol loaded
Daytime protocol loaded
SMTP protocol loaded
POP protocol loaded
IMAP protocol loaded
HTTP protocol loaded
FTP protocol loaded
NNTP protocol loaded
Component: "System Port" 1.1.2.5 (1-Feb-2003/6:43:38)
Script: "User Preferences" (10-Dec-2002/18:35:10+1:00)
Script: "Prebol - Official REBOL Preprocessor" (none)
Missing input file name argument
>> do/args %prebol.r [ %main.r %421.r ]
Script: "Prebol - Official REBOL Preprocessor" (none)
Processed: 12223 bytes
== none
>>
```

Figure 5-12. *Using Rebol's preprocessor.*

The SDK comes with a number of different programs to encapsulate a script. (That is to create an executable binary from the script). These automatically call `prerebol` whenever a script is transformed into a binary. The use of `prerebol` through commands is not necessary when you test your projects in the different Rebol environments (Rebol/base, Rebol/face, Rebol/cmd, etc.).


REBOL/SDK – Preprocessor
file:///Users/olivier/Desktop/SDK/rebol-sdk-cmd/docs/prebol.html
Google
Apple France
.Mac
Amazon France
eBay France
Yahoo!
Informations
REBOL
Documentation | User's Guide | Dictionary | Library | Feedback

REBOL/SDK - Preprocessor

Return to Contents

Contents:

1. Overview
    1.1 Part of Encap
    1.2 Preprocessor Source Code
    1.3 Changes in 2.0
2. Running the Preprocessor Separately
3. Preprocessor Commands
    3.1 Include Commands
    3.2 Including Groups of Files
    3.3 Include File Headers
    3.4 Evaluation Commands
    3.5 Conditional Includes
4. Example Usage
5. Hint for Creating Module Namespaces
6. Including Scripts Not to be Preprocessed

1. Overview

This tool allows you to combine multiple source or data files into a single REBOL output file.

The preprocessor includes the full power of REBOL, allowing you to evaluate any REBOL expression during processing. The program allows merging of code, data, images, sound and other types of files into a single output file. It also supports conditional expressions, such as the conditional including of files.

The output is a reduced REBOL source file with extra spacing, comments, and include file headers removed. Processing is recursive, so included files can themselves contain commands to process. This is very powerful.

Figure 5-13. *The preprocessor documentation.*

If you do not work with Rebol/SDK but use the free versions of Rebol (Core and View), you can simply download the `prebol.r` from www.reboltech.com/library/scripts. Then you get a Rebol script performing the same functions as the commercial version. The usage is the same and so are the results. Please note that the commercial version is version number 2.0 and is lighter and faster than the free one, version 1.0. In fact, the differences are not noticeable to the visible eye and all the examples that follow are based on the free version of the preprocessor.

Including files

To illustrate the use of the Rebol preprocessor, you are going to develop a script, very freely inspired by the game of 421. It is to roll three dice to obtain the values 4, 2, 1. This program, developed with Rebol/View, is composed of two scripts: the main programme called `main.r` and a library named `launch.r`. The application is also built with some graphics files in PNG format and two text files to facilitate the localisation of the product in English or French.

The various functions of the program are in the `main.r` file. These include all the resources necessary to build a single working script, called 421.r, which can be easily distributed over the Internet.

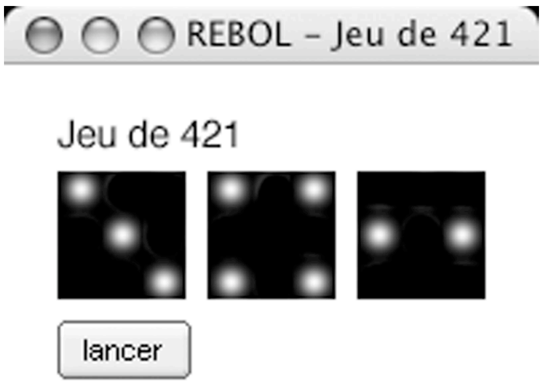


Figure 5-14. *The 421 game.*

To include data, Prebol has four commands which are `#include`, `#include-binary`, `#include-files` and `#include-string`. These instructions take a file type (`file!`) parameter or, in the case of `include-files`, a directory and a block containing file names. It is also possible to use Rebol expressions enclosed in brackets. In that case, the result of the evaluation is used as the parameter.

To include the `launch.r` library, you simply insert the command, `#include %launch.r`, in your script. Images are inserted with the `#include-binary` or `#include-files` commands. Obviously, you can use other types of data such as sound files. In fact, these commands simply insert binary data, whatever the format. For the 421 project, the images are the application title (`titre.png`) and six images for the different faces of a dice. These files are stored in a sub-directory called `pics` in the directory which contains the project. The title is imported with the statement `titre.png: load #include-binary (join %pics/ %titre.png)`. Here we used an expression which will be evaluated by Rebol to build the file path. When the preprocessor is used, the contents of binary files are converted to base 64 and inserted in the generated file. When evaluating the script, the data will be loaded in the word `titre.png`.

For the different faces of the dice, you are going to use the `#include-files` directive. It allows you to include several files from a directory in a single operation.

```
#include-files %pics [
      cube1.png
      cube2.png
      cube3.png
      cube4.png
      cube5.png
      cube6.png
]
```

For each file in the list, this command produces a pair of values composed of the file name (resource identifier) and the file contents. To automatically create words containing data, you can use the `foreach` instruction which allows you to traverse the series and assign all values to words.

We must use the result of the preprocessor to recover each identifier and its corresponding value. The resources are easily added to the Rebol dictionary with `set`.

```
foreach [ word data ] #include-files %pics [
      cube1.png
      cube2.png
      cube3.png
      cube4.png
      cube5.png
      cube6.png
] [ set word load data ]
```

Evaluation and conditions

To localise the application, you'll have to conditionally include a file with the button titles. If you generate the French version, it will be the `fr.txt` file. If the product is going to be in English, it will be `en.txt`. To indicate the language to be used to the preprocessor, you must use a variable which will be evaluated as the product is built. For this you have the `#do` command.

This instruction is very powerful as it allows the execution of Rebol code from Prebol. It allows parameters to be set, files to be deleted, data to be sent to a FTP or HTTP server, a file journal to be constructed, etc.

For your project, you are going to use this command to simply initialise the variable `lang` with the value “fr” or “en”.

```
#do [ lang: "fr" ]
```

For conditions, you have both the commands `#if` and `#either.` They work on the same principles as the words `if` and `either` of Rebol. A block is evaluated and if the result is `true`, the following block is executed. The `#either` instruction adds a third block which is only evaluated if the value returned is `false`. In your project, you are going to set the word `launch-txt`, which is the button title, to the character string contained in the `fr.txt` or the `en.txt` file.

```
launch-txt: #either [lang = "fr" ] [
      #include-string %fr.txt
] [ #include-string %en.txt ]
```

All that remains is to generate your script with the help of Prebol. You will get a file of around 16k bytes containing your application code and the resources it needs to work. This single file can now be broadcast over the Internet.

Summary

Rebol is versatile language perfectly suitable for writing network applications or office software. Rebol/Command is dedicated to *e-Business*. Rebol/IOS is a *groupware* platform that astonishes with its ease of use and possible extension. It is also possible to use the free versions of Rebol to develop web applications.

6
Rebol for geeks

Rebol is a fascinating language which allows the use of original and efficient technologies with breathtaking ease. Deploying virtual desktops over the Internet and programming Unix and Windows applications are commonplace activities for the Rebol programmer.

Rebol and virtual desktops

You can build virtual offices to disseminate information or applications over the Internet or your company intranet with Rebol/View. It is one of the major revolutions introduced by Rebol.

A work environment

With Rebol/View, each user can access a connection point giving entry to private or public services. These are Rebol programs downloaded over the network from an HTTP server: they are known as Reblets.

Each person in a company or organisation, either static or nomadic, can have a range of services without the need to install applications on the client machine. A virtual office can not only provide access to data files (text, XML documents, etc.) but also to classic web services (local or remote HTML documents, intranet applications, etc.). Again, the tool is innovative yet flexible enough to fit into an existing architecture. It isn't necessary to change everything at once, your intranet's different services can be progressively migrated to a Rebol/View virtual office which then gradually becomes your work environment.

The Rebol/View desktop organisation

At the top of the Rebol/View screen, you will find shortcuts such as "User" to set the configuration for the user or "Goto" to access a virtual office via its URL. These various headings can be customised through the `services.r` file in the `desktop` directory of the Rebol/View file tree. At the bottom of the window, you will find an area designed to display status information. On the left, the Rebol/View console is accessible by a simple click of the mouse button. There is also an icon, with the default settings, for the `Rebol.com` directory which opens the doors of the `World Wide Reb`. You can add further shortcuts to this window pane by editing the `bookmarks.r` file which is also in the `desktop` folder. As an exercise, edit this file with a simple text editor and give it the following content :

```
REBOL [Title: "Bookmarks" Type: 'index]
folder "REBOL.com" http://www.rebol.com/index.r
folder "Local site" %local/index.r
folder "rebsite" http://172.29.143.1/rebsite/index.r
file "Console" console icon console
```

Now on launching Rebol/View, two new directories will appear. The "Local site" folder allows you to access files in the local Rebol/View directory. The second is a link to a rebsite that you're going to put in place on an http server whose IP address is 172.29.143.1.

At the bottom right of the window, you will find the current mode of working of Rebol/View. If it is "local" mode, you click on it to change to connected mode. Otherwise your desktop will be unable to communicate with the server.

The file index

You just store the files that make up your virtual office on the HTTP server. The file structure is defined by the files named `index.r`. In each sub-directory there is an `index.r` file which defines the content of that directory. These files should include in their Rebol header that they are of the `'index` type. As an exercise, you can create an `index.r` file indicating the name of your desktop and the links available. A welcome text message is contained in the file called `welcome.txt`. One entry loads a reblet, another links to a website's homepage. In this last case, Rebol/View displays the HTML document with the help of the machine's default browser. A GIF image is used to "decorate the office" but many other effects are possible.

```
REBOL [type: 'index]
title "My rebsite"
backdrop %fond.gif
file "Welcome" %welcome.txt
file "A Reblet" %reblet.r info "an example Reblet"
link "My Website" http://www.mywebsite.org
```

The reblet is a tiny script which displays some text in a window :

```
REBOL []
view/title layout [ title "A Reblet !" ] "A Reblet"
```

Using a virtual office, you can actually boost your intranet and transform it into an intelligent, interactive tool.

Unix programming with Rebol

Rebol is present on many platforms, most of whom are members of the great Unix family.

The Core, View and Command interpreters are available for Linux, FreeBSD, OpenBSD, Solaris and Mac OS X. These systems have many features that Rebol is quite capable of exploiting to produce high-performance applications.

This section is devoted to Unix programming using Rebol. We are going to discover how to effectively display in a console, manage the keyboard, interface a Rebol application with the shell, obtain and define file access rights, and to communicate through pipes and sockets.

Displaying data in a console

In the Unix world, three types of interface can be seen by the user. Under X-Windows, applications can use modern graphic components such as windows and buttons with the help of a mouse. Other programs, such as webmin, run in a web browser; their interfaces are built in HTML. Developing both these types of software is easy with Rebol. The View evaluator has a powerful dialect called VID which allows for the construction of advanced graphical applications with minimal code. For web applications, Rebol supports standard CGI, FastCGI and handles tags and transmission settings with ease. These two aspects of developing with Rebol have already been covered in this book. They are not solely related to the Unix platform and, for that reason, are not covered in this chapter.

That leaves the third type of user interface, the good old, reliable text mode with a simple console or simple telnet client. One might think that this type of software has been totally surpassed but, on Unix, this is not yet the case. Many utilities and even some heavy management applications still use character based interfaces. These provide many advantages such as speed of display, the reduced cost and low maintenance terminal, low consumption of network bandwidth and economical use of the server's processing power.

To display character strings on the screen, Rebol provides the words `print` and `prin`. The only difference between them is that `print` adds a newline after the data has been displayed. These two words take a single argument that can be of different types. If the argument is a list, its contents are automatically evaluated before they are displayed.

Rebol also includes different control characters such as `#"^/'`" to insert a carriage return or `#"^(tab)"` for a tab character. It is also possible to change the width of a tab with the help of the property `system/console/tab-size`.

The following example displays a person's first and last names on two lines using 8 character tabs:

```
system/console/tab-size: 8
last-name: "Bridge"
first-name: "Peter"
print [ "Name:^(tab)" last-name "^/First Name:^(tab)" first-name ]
```

Happily for us, Rebol's display capabilities are not limited to just this. To embellish the presentation of your applications, Rebol allows the display of character sequences that will be interpreted by the terminal. These commands can clear the screen, move the cursor, change the text's appearance and even use colour. These character sequences all begin with the ASCII character code 27 followed by the "[" character, or in Rebol `"^(1B)["`.

Knowing this, it is possible for us to clear the console screen, position the cursor at the tenth column of the fourth line and display a short message:

```
ESC: "^(1B)["
prin join ESC "J"
prin join ESC [ "4;10H"]
prin "Hello !"
```

By using the sequence "`7n`", you can also get the number of rows and columns in the console. This involves creating a port using the built-in `'console` protocol.

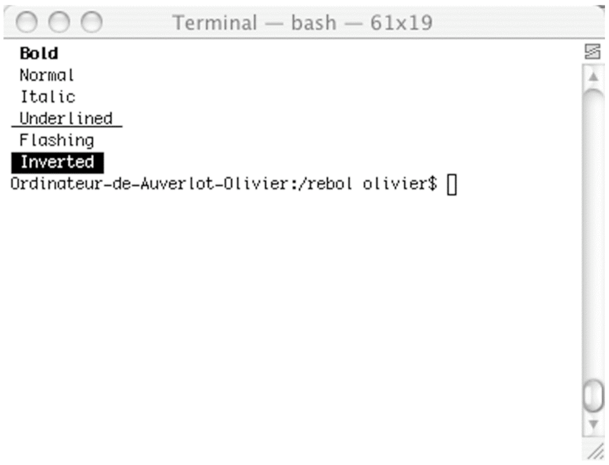
Once the sequence is sent, reading the third element of this port returns a result formatted with the number of lines separated from the number of columns by a ";" (semi-colon) and the string is terminated with a capital "R".

```
cons: open/no-wait/binary [ scheme: 'console ]
print "^(1B)[7n"
pos: parse next next to-string copy cons ";R"
close cons
print make pair! reduce [
      (to-integer second pos) (to-integer first pos)
]
```

You can easily put in place an advanced interface using these control sequences. You can also change the appearance of text and use bold, underlined, italic and even flashing characters.

To make menus or input fields, it is possible to reverse the text and background colours using the code "`7M`". The following example displays the different effects that are possible. If the use of these sequences pose no technical difficulties, keep in mind that the results may differ depending on the capability of the terminal being used.

```
ESC: "^(1B)["

styles: [
      [ "Bold" "1m" ]
      [ "Normal" "2m" ]
      [ "Italic" "3m" ]
      [ "Underlined" "4m" ]
      [ "Flashing" "5m" ]
      [ "Inverted" "7m" ]
]

prin join ESC "J"   ; clear the screen

foreach ele styles [
      print [ (join ESC second ele) (first ele) (join ESC "0m") ]
]
```



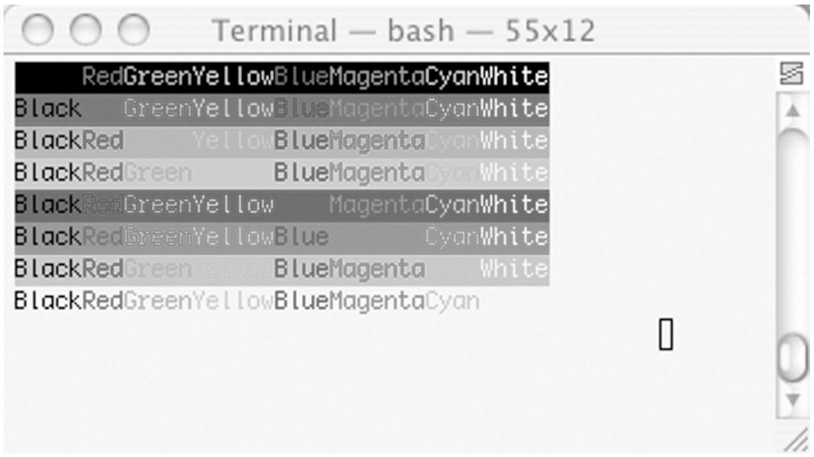
```
Bold
Normal
Italic
Underlined
Flashing
Inverted
Ordinateur-de-Auverlot-Olivier:/rebol olivier$
```

Figure 6-1. The various control sequence effects.

To further improve the appearance of your applications, you can choose from a list of eight predefined colours (black, red, green, yellow, blue, magenta, cyan and white).

For each of them, you must use a specific character sequence that differs depending on whether you want to change the colour of the text or its background. The following example illustrates the different possibilities you can achieve by displaying the different available colours.

```
ESC: "^(1B)["

text-colour: [
      [ "Black" "30m" ] [ "Red" "31m" ] [ "Green" "32m" ]
      [ "Yellow" "33m" ] [ "Blue" "34m" ] [ "Magenta" "35m" ]
      [ "Cyan" "36m" ] [ "White" "37m" ]
]

background-colour: [
      [ "Black" "40m" ] [ "Red" "41m" ] [ "Green" "42m" ]
      [ "Yellow" "43m" ] [ "Blue" "44m" ] [ "Magenta" "45m" ]
      [ "Cyan" "46m" ] [ "White" "47m" ]
]

prin join ESC "J"   ; Clear the screen

foreach bc background-colour [
      prin join ESC (second bc)
      foreach tc text-colour [
            prin join ESC [ (second tc) (first tc) ]
      ]
      prin #"^/"  ; carriage return
]
```



```
     RedGreenYellowBlueMagentaCyanWhite
Black     GreenYellowBlueMagentaCyanWhite
BlackRed       YellowBlueMagentaCyanWhite
BlackRedGreen       BlueMagentaCyanWhite
BlackRedGreenYellow      MagentaCyanWhite
BlackRedGreenYellowBlue         CyanWhite
BlackRedGreenYellowBlueMagenta      White
BlackRedGreenYellowBlueMagentaCyan
```

Figure 6-2. *The text and background colours can be changed.*

If the screens of your application are complex and require numerous operations to be displayed, it is necessary to optimise them.

In terms of speed, the easiest gain is obtained by avoiding the use of the `print` and `prin` words for each control sequence sent to the screen..

The idea is to build the screen in a buffer and to display its contents with a single call of the `print` word. The following example illustrates this method by demonstrating an improved version of the previous example:

```
buffer: copy ""
insert tail buffer join ESC "J"   ; clear the screen
foreach bc background-colour [
      insert tail buffer join ESC (second bc)
      foreach tc text-colour [
            insert tail buffer join ESC [ (second tc) (first tc) ]
      ]
      insert tail buffer #"^/"     ; carriage return
]
prin buffer
```

Another very logical approach that is extremely effective is to only update those parts of the screen whose contents have been changed. By avoiding the automatic redisplay of the whole screen for each change, your application will be more responsive and may even create less network traffic.

Managing the keyboard

We now know how to display information. The next step is to make our applications react to keyboard actions. For keyboard entry captured by the "Return" key, Rebol provides the `input` and `ask` words which wait for the entry of characters from the standard input peripheral (stdin).

These two words have a *refinement* `/hide` for capturing confidential information (an asterisk is displayed instead of the character pressed). `ask` takes an argument that is a question to be posed to the user.

```
print "What is your last name:"
last-name: input
first-name: ask "What is your first name:"
mdp: ask/hide "Your password? "
```

The word `confirm` requests a user to confirm an action.

By default, the response must be “Y” or “N” but the *refinement* `/with` allows the specification of two characters to symbolise acceptance or refusal. The result returned is a boolean value.

```
confirm/with "(A)ccept or (D)ecline " [ "a" "d" ]
```

To supplement these simple functions, the Rebol interpreter offers other features to manage the keyboard at a lower level. For instance, it is possible to disable the ESC key to restrain a user from stopping execution of an application. For this, all you need to do is to set the `break` property of the `system/console` object to `false`. You can also manage the main control keys on the keyboard. Take care though, all keyboards are not identical (try to find the Mac “apple” key on a PC!). For this reason, the Rebol interpreter identifies a group of keys common to the various operating systems under which it runs. In addition to displayable characters, the console recognises the enter, tab, del, up, down, right, left and insert keys.

To manage your keyboard more precisely, you must use the `‘console` protocol to get the code from the actual keys when pressed. Once the port is open, the application can enter an infinite loop and wait for a key to be pressed with the help of the word `input?` Which returns the boolean value `true` if a new keystroke has been captured. Reading the port retrieves a string whose length depends on which key was pressed. If the sequence does not start with the control character ^(1B), it is a single character or a special key such as tab, enter or del. If the sequence starts with ^(1B) and the second character is also ^(1B), the script has detected the use of the ESC key. The third and last case is the use of the insert, up, end and the arrow keys. The following script illustrates this type of keyboard management and can easily be adapted to any type of project.

```
system/console/break: false
cons: open/no-wait/binary [ scheme: 'console ]
forever [
      until [ input? ]
      touch: copy cons
      either (first touch) <> #"^(1B)" [
            switch/default (first touch) [
                  8 [ print "DEL" ]
                  9 [ print "TAB" ]
                  13 [ print "ENTER" ]
                  127 [ print "DEL" ]
            ] [ print to-char first touch ]
      ] [
            either (second touch) = 27 [
                  print "ESC"
            ] [
                  switch (third touch) [
                        50 [ print "INSERT" ]
                        65 [ print "UP" ]
                        66 [ print "DOWN" ]
                        67 [ print "RIGHT" ]
                        68 [ print "LEFT" ]
                        101 [ print "END" ]
                  ]
            ]
      ]
]
close cons
```

Rebol assures your code is compatible across different platforms by supporting a limited number of keystrokes. Sticking to this selection guarantees that your applications work perfectly on the numerous systems on which Rebol runs.

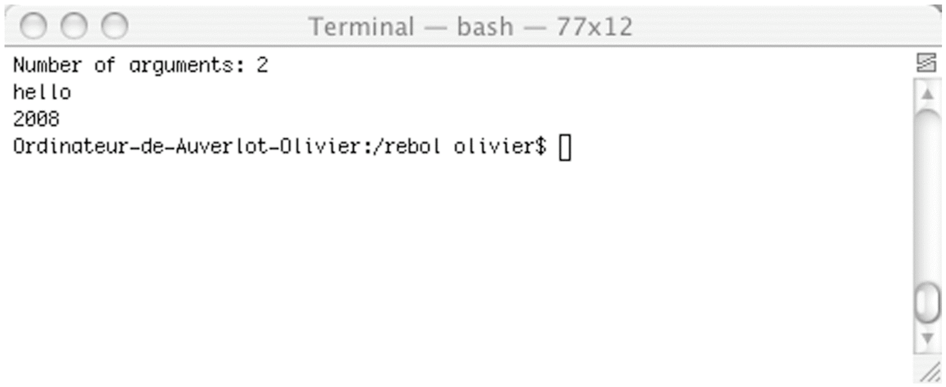
Integration with the Unix environment

Rebol applications fit perfectly into the Unix environment and during their execution nothing distinguishes them from applications written in the most popular Unix languages such as C or Perl. For a Rebol script to be recognised as an executable program, you must insert a shebang as first line in its file. (That is the "#!" directive followed by the access path of the Rebol interpreter and the arguments you want to pass to Rebol). The arguments normally passed to Rebol are usually `-q` and `-s` to prevent Rebol from displaying its start-up information and to disable the built-in security manager. You must also set the execution rights of your script with the classic `chmod +x`.

Unix applications often receive arguments passed from the command line. With Rebol, these are stored in the `options` object of the `system` object. The `args` property contains `none` if no command line arguments were provided. In the other case, it contains a list of n elements. To find the number of parameters, you simply use the word `length?` Which returns the length of a value.

The following example verifies the possible presence of arguments passed to the script and, if there are, display the number of arguments and their respective values.

```
if not none? system/options/args [
      print [
            "Number of arguments:"
            (length? system/options/args)
      ]
      foreach arg system/options/args [ print arg ]
]
```



```
Number of arguments: 2
hello
2008
Ordinateur-de-Auverlot-Olivier:/rebol olivier$
```

Figure 6-3. Arguments passed from the command line.

With the help of the `system/options` object, you can obtain precise information about the environment in which your application is running. The `path` property returns the current directory. With `home`, you can determine the directory access path of the user. Finally, the `boot` property contains the location of the Rebol interpreter in the file system.

Accessing system environment variables is possible by using the `get-env` word. It takes a single parameter which is the name of the desired environment variable. So the syntax `get-env "SHELL"` returns the path to the shell. Since you're on Unix, don't forget that you must strictly adhere to the correct upper and lower case characters of the environment variable names. If `get-env` can't provide a result, it returns the `none` value.

The redirection of the standard output device (stdout) to a file is also possible with the help of the word `echo`. This sends the characters displayed on the screen to a file whose name is passed as a parameter. In Rebol, file and filepath names must start with the "%" character.

So, if you want to save the screen display in a file called test.txt in the directory /var/tmp, all you need to do is to insert the command echo %/var/tmp/test.txt in your application. From now on, all the data displayed on the screen will be saved in the file test.txt. To deactivate this redirection, you use the syntax echo none.

Files and access rights

Managing files is one of the principal activities of Unix programmers.

By using the word info?, you can find the last modification date of a file, its size in bytes and whether it is a file or a directory. This word returns a simple object composed of three properties which are size, date and type.

To get this information about the file test.txt, you simply use print mold info? %test.txt.

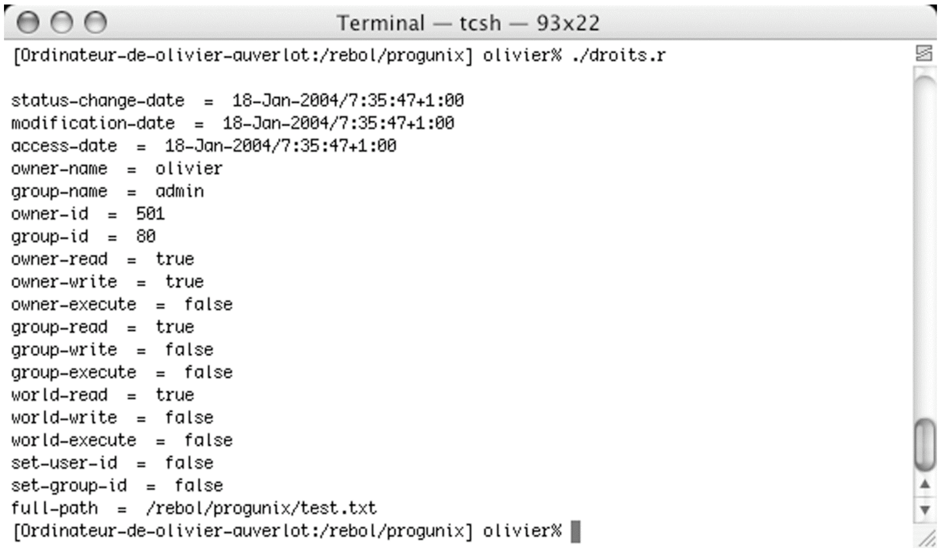
Access to various elements of the Unix file tree is regulated through access rights which can be determined through the word get-modes. Rebol programmers have access to this through a port pointing at the resource being studied.

The following example retrieves the name of the owner of the file test.txt by using 'owner-name as the argument.

```
p: open %test.txt
print get-modes p 'owner-name
close p
```

The different attributes can be found by using the value 'file-modes as the argument of get-modes. The list obtained is extremely comprehensive as you have rights over the file for the owner, for their group and for other users. The information also contains the name of the owner, their group, UID (*User ID*), GID (*Group ID*) and even the complete access path to the file. The following script displays all the properties of the file test.txt.

```
p: open %test.txt
d: get-modes p 'file-modes

forall d [
      print [ (first d) " = " get-modes p (first d) ]
]

close p
```



```
[Ordinateur-de-olivier-auverlot:/rebol/progunix] olivier% ./droits.r

status-change-date  =  18-Jan-2004/7:35:47+1:00
modification-date  =  18-Jan-2004/7:35:47+1:00
access-date  =  18-Jan-2004/7:35:47+1:00
owner-name  =  olivier
group-name  =  admin
owner-id  =  501
group-id  =  80
owner-read  =  true
owner-write  =  true
owner-execute  =  false
group-read  =  true
group-write  =  false
group-execute  =  false
world-read  =  true
world-write  =  false
world-execute  =  false
set-user-id  =  false
set-group-id  =  false
full-path  =  /rebol/progunix/test.txt
[Ordinateur-de-olivier-auverlot:/rebol/progunix] olivier%
```

Figure 6-4. *Rebol allows the consultation and modification of access rights .*

With `set-modes`, you can modify the rights of a file. This words uses two parameters which are the port pointing at the file to be manipulated and a block containing the attributes to be modified or added.

The following example modifies the access rights of the file `test.txt` by giving write access to members of the owner's group and other users.

```
p: open %test.txt
set-modes p [
      group-write: true
      world-write: true
]
close p
```

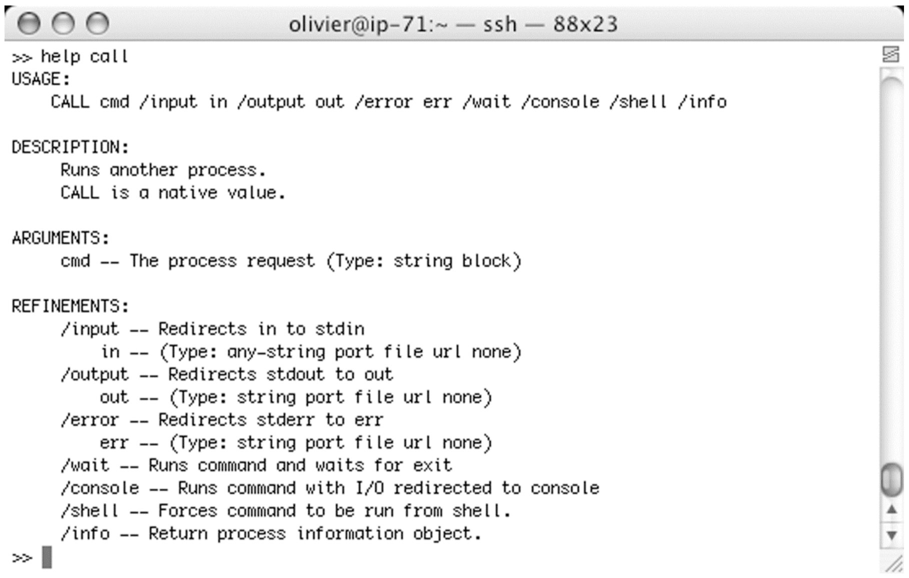
Through the use of a port to handle file access rights, Rebol provides an interface with a high level of abstraction compared to the operating system and hardware platform.

The programmer is not penalised in any way by this architecture which leaves him or her in total control of operations and allowing them to easily access the power of Unix system access rights.

Here Rebol remains true to itself by demonstrating that a tool can remain simple to use.

Shell access

If you wish to launch external applications such as the powerful Unix commands, you must use the shell of the operating system. This feature has always been available in the commercial versions of rebol but has also been included in the latest free versions. In Rebol, it's the word `call` which allows dialogue with the shell. It takes a single argument which is a string of the path of executable files to be launched. The parameter is directly transmitted to the shell, the syntax of this path must conform to the standards of the host platform and not to those of the Rebol interpreter. For example, the command `call "ls -l"` displays a list of the files in the current directory. Likewise, to call the executable test stored in `/usr/local/bin`, the syntax is `call "/usr/local/bin/test"`.



```
>> help call
USAGE:
    CALL cmd /input in /output out /error err /wait /console /shell /info

DESCRIPTION:
     Runs another process.
     CALL is a native value.

ARGUMENTS:
     cmd -- The process request (Type: string block)

REFINEMENTS:
     /input -- Redirects in to stdin
         in -- (Type: any-string port file url none)
     /output -- Redirects stdout to out
         out -- (Type: string port file url none)
     /error -- Redirects stderr to err
         err -- (Type: string port file url none)
     /wait -- Runs command and waits for exit
     /console -- Runs command with I/O redirected to console
     /shell -- Forces command to be run from shell.
     /info -- Return process information object.
>> 
```

Figure 6-5. *Call controls the execution of external applications.*

There are many *refinements* that make this word a very flexible tool. With `/info`, the word `call` returns an object whose property `id` is the PID of the process launched. To capture the display of the shell, you use the `/console` *refinement*.

Synchronising tasks is also possible with the `/wait` *refinement* which obliges the interpreter to wait until the process launched finishes before continuing to process the script. Finally, the *refinements* `/input` and `/output` authorise the use of Unix redirections. Assuming that you want to count the number of lines in a file with the Unix command `wc`, the Rebol syntax will be `call/input "wc -l" %test.txt` (which corresponds to `wc -l < test.txt`). To redirect data written to the standard output device (stdout) to a Rebol variable, you only need to use `/output`. For example, redirecting and storing the results of the files present in the current directory in a variable `buf` you write `call/output "ls -l" buf`. Obviously, these refinements can be used together to exploit the tremendous potential of the Unix shell.

Inter-process communication

Exchanging data between applications is a fundamental part of Unix programming. Rebol allows these operations with the help of anonymous pipes and sockets.

Pipes are unidirectional communications mechanisms. Defined by the `S_IFIFO` type in POSIX, A pipe is simply a file system node which is composed of two separate entries in the file table. These entries can be both read and written. Two Unix processes can exchange information according to a sender-receiver model. The dataflow is a continuous flow of characters since the receiver cannot distinguish between the different transmissions from the sender. Moreover, the first data inserted into a pipe are always the first to be read. Once the data is read, the information collected is removed from the pipe to free traffic along it. It is therefore not possible to have direct access to information contained in the pipe: the data must be stored by the recipient before being processed.

To read and write data in an anonymous pipe, Rebol provides `input` and `output` ports of the `system/ports` object. Information is read by sampling data in `system/ports/input` as it becomes available. So, if we want our Rebol script `myscript.r` to read and display the data produced by the Unix command `ls -l` (using the syntax `ls -l|myscript.r`), the code needed is a simple loop reading the `input` port:

```
while [
      my-line: pick system/ports/input 1
] [ print my-line ]
```

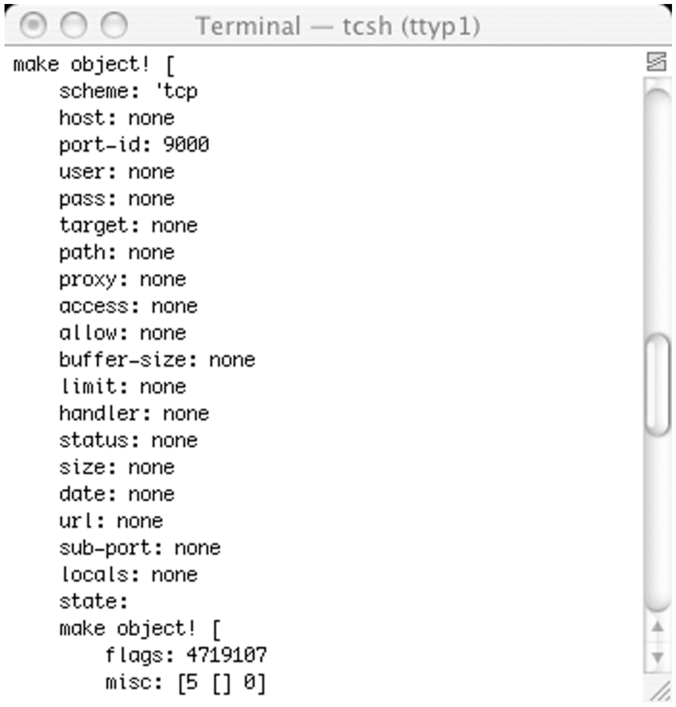
Writing data to an anonymous pipe is hardly more difficult then reading from one. A Rebol script can transmit data to another Unix application with great ease. All that is needed is to insert the data in the `system/ports/output` port. The following example illustrates this through the use of the Unix instruction`./myscript.r|more`. The latter has the effect of displaying the data passed through a pipe on the screen:

```
data: {
      a message transmitted
      via an anonymous pipe
}
insert system/ports/output data
```

If anonymous pipes are a simple and effective way to transmit data between applications, they remain limited to exchanging data between programs running on the same machine. To establish inter-process communications based on a distributed architecture, Unix systems offer an elegant mechanism which uses the TCP/IP protocols and which is referred to by the term socket. Introduced in Berkeley Unix distributions, sockets are two-way points of communication by which a process can send and receive data. As a dedicated network language, Rebol considerably reduces the work of the programmer in writing servers and clients which use the TCP or UDP protocols.

Creating a TCP server begins with declaring the listening socket to be used to receive connection requests. In Rebol, a socket is represented by an object of the type `port!` and is created by using `open` followed by a URL. The latter contains the port number used to receive client connections.

To study the structure of a socket and to learn about the different properties available, all you have to do is use the syntax `print mold` followed by the name of the socket.



```
make object! [
    scheme: 'tcp
    host: none
    port-id: 9000
    user: none
    pass: none
    target: none
    path: none
    proxy: none
    access: none
    allow: none
    buffer-size: none
    limit: none
    handler: none
    status: none
    size: none
    date: none
    url: none
    sub-port: none
    locals: none
    state:
    make object! [
        flags: 4719107
        misc: [5 [] 0]
```

Figure 6-6. *A TCP socket is represented by a object of the type port!.*

Once the socket is opened, the script can then enter an infinite loop and wait for requests from clients by using the word `wait`. Once a contact is established, the client socket is returned as the first element of the listening socket and data received are recovered with the help of the word `read-io`. Responses from the server are transmitted to the client by inserting the information into the connected socket. The latter may then be released with the word `close`. The following example is a small TCP server which listens on port 9000. It receives a character string, reverse its contents and return it to the client. A carriage return is used by the client to signify that the transmission of the string is finished.

```
p: open tcp://:9000
forever [
      wait p
      conn: first p
      buffer: copy ""
      until [
            data: copy ""
            read-io conn data 255
            append buffer data
            found? find data "^/"
      ]
      insert conn (head reverse buffer)
      close conn
]
```

The work of the client begins with declaring a socket with the help of a URL containing the name or IP address of the machine hosting the server and the port on which the server is listening. Again, Rebol considers the socket to be an object of the type `port!`. Sending data to the server is simply achieved by inserting the information into the port and reading it is done with `copy`. Connections are closed with the word `close`. The following example is a client for the server shown above. The user types a string of characters which are transmitted to the server. The response from the server is displayed upon the screen before the connection is closed.

```
forever [
      p: open tcp://localhost:9000
      txt: ask "#"
      insert p join txt "^/"
      print copy p
      close p
]
```

Generally, it is the TCP protocol which is used to exchange data between processes. It provides a reliable data transport mechanism. If speed is the determining criteria, Rebol also supports the UDP protocol who use is practically identical except for the description of the socket. This protocol also makes it possible to use *broadcast* and *multicast* to deliver information to multiple machines.

Finally, remember that Rebol incorporates many different high-level protocols such as HTTP, SMTP and FTP and it is even possible to create your own protocols with the help of the root object called `root-protocol`.

Managing Unix signals

By using sockets with Rebol you can quickly develop background server processes. These *daemons* generally need to communicate both with the system and users in order to be able to receive commands such as “stop” or “restart”.

To achieve such operations, Unix systems offer an ingenious mechanism known by the term *signal*. You have access to a set of signals that can be issued by one process to another which either have a predefined function or one that is left to the discretion of the programmer.

A signal is identified by a positive number and a unique symbolic name. An event is associated with each of them but a signal may well be transmitted to another process without the event being produced. The different signals available on a UNIX system are listed in the `signal.h` header file of the C compiler for the machine.

Signal	Number	Description
SIGHUP	1	Problem on the terminal or stop a child process by the parent process.
SIGINT	2	The process was stopped by the use of the ctrl+c key combination.
SIGQUIT	3	Identical to a SIGINT except save the memory state in the current directory.
SIGTERM	15	Stop the process.
SIGUSR1	16	User definable signal.
SIGUSR2	17	User definable signal.

List of managed signals.

Signals are an important aspect for developing under Unix. In effect, they can warn an application that a significant event has occurred, the application must react appropriately.

For example, if the ctrl + c key sequence is pressed by the user to stop execution of the active program, then it must properly closedown saving any data that it was processing. *Daemons* must generally respond to orders transmitted via Unix commands such as `kill` or `signal`. If the order is to stop or restart, the program must be able to properly execute the order that it was sent.

Therefore managing signals in an application requires setting up an event manager which reacts to signals received.

In Rebol, you have at your disposal a dedicated protocol called `system` (not to be confused with the `system` object which has quite a different role). The protocol allows Rebol to handle the arrival of the `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGUSR1` and `SIGUSR2` signals. In order to use this protocol a `system` port must be opened with the word `open`. Once this has been completed, you can define a filter to specify signals to be intercepted by an event handler. The word `set-modes` takes two parameters which are the port being used and a block in which the signal property is present. This property receives a value which is a block containing the different signal traits. If all signals are to be handled, the `get-modes` returns a list of them in the word `‘signal-names`.

Once the filter is defined, you can now add this event handler to those already present in Rebol. Simply add the port that handles signals into the block `system/ports/wait-list`. The function `enable-system-trap` carries out these different operations and activates the signal listener in the Rebol interpreter.

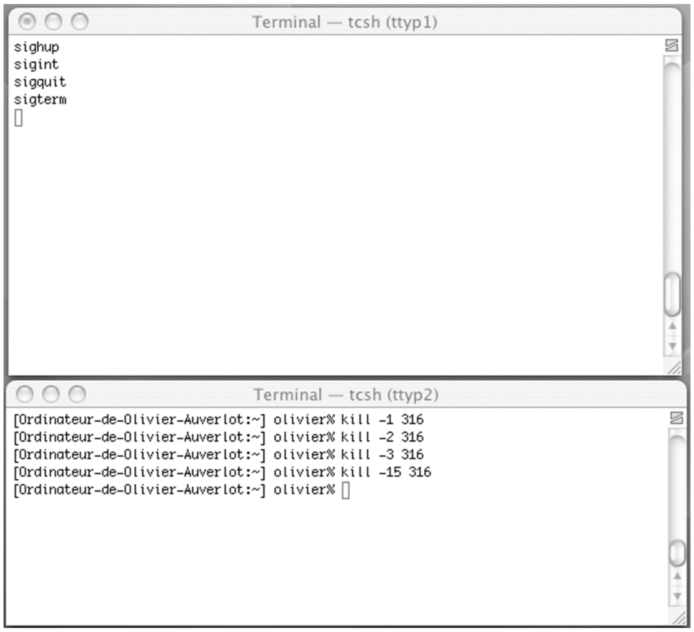
```
enable-system-trap: does [
      system/ports/system: open [ scheme: 'system ]
      set-modes system/ports/system [
      signal: get-modes system/ports/system 'signal-names
 ]

append system/ports/wait-list system/ports/system
```

To find out if a signal has been trapped by your Rebol script, all you have to do is periodically check the contents of the `system/ports/system` port.

If a signal has arrived, the message received will be a block composed of two elements which are the word `'signal` and the name of the signal.

The function `check-system-trap` performs this operation and displays the name of the signal captured on the screen:

```
check-system-trap: func [ /local msg ] [
      while [ msg: pick system/ports/system 1 ] [
            print form second msg
      ]
]
enable-system-trap

forever [ check-system-trap ]
```



```
sighup
sigint
sigquit
sigterm
```

```
[Ordinateur-de-Olivier-Auverlot:~] olivier% kill -1 316
[Ordinateur-de-Olivier-Auverlot:~] olivier% kill -2 316
[Ordinateur-de-Olivier-Auverlot:~] olivier% kill -3 316
[Ordinateur-de-Olivier-Auverlot:~] olivier% kill -15 316
[Ordinateur-de-Olivier-Auverlot:~] olivier%
```

Figure 6-7. *The Rebol script detects signals transmitted by the Unix kill command.*

Handling signals demonstrates the ease with which the Rebol interpreter is able to communicate with its host's run-time system. Such ease is present when a Rebol script interfaces with functions and data structures in dynamic libraries.

Interfacing with dynamic libraries

Previously only Rebol/IOS, Rebol/View/Pro, Rebol/SDK and Rebol/Command could interface with native code dynamic libraries. Since the release of Rebol 2.7.6, Rebol/View can also access external dynamic libraries. This means Rebol scripts can declare and call functions written in languages such as C or C++. Some parts of scripts can be optimised by calling native functions whose execution speed is much faster than interpreted Rebol code. Rebol can not only take advantage of features in the operating system API but also enjoy the many specialised libraries (image generation, producing pdf documents, connecting to database engines, etc.). Such an library is used so that Rebol can connect to use the Berkeley DB (http://www.cs.unm.edu/~whip) and DyBase database engines (http://www.garret.ru/~knizhnik/dybase.html).

Rebol includes everything that is necessary to easily and quickly write library wrappers. The first step is to load the dynamic library whose functions will be used. This operation is performed with the word `load` with its `/library` *refinement* which returns an identifier. Declaring the different functions is done using the `routine!` datatype with the declaration of the function's input parameters, its return value, the identifier of the library containing the function and the name of the function within the dynamic library. The word obtained is added to the dictionary and has the same attributes as all other words. You can even provide documentation so that the `help` word will provide information about its use. The only difference being, recognising that it is a native function, its source code is not visible. Finally, when the dynamic library is no longer needed, you can reclaim the memory it occupied by using the word `free` followed by the name of its identifier.

To better understand how this works, we will add the word `get-nprocs` to the Rebol dictionary. It will use a function in the standard `libc` library to find the number of processors on a machine. This function doesn't take any input parameters and returns only a simple integer.

```
lib-so: load/library %libc.so.6

get-nprocs: make routine! [
      return: [ integer! ]
] lib-so "get_nprocs"

print [ "Number of processors:" get-nprocs ]

free lib-so
```

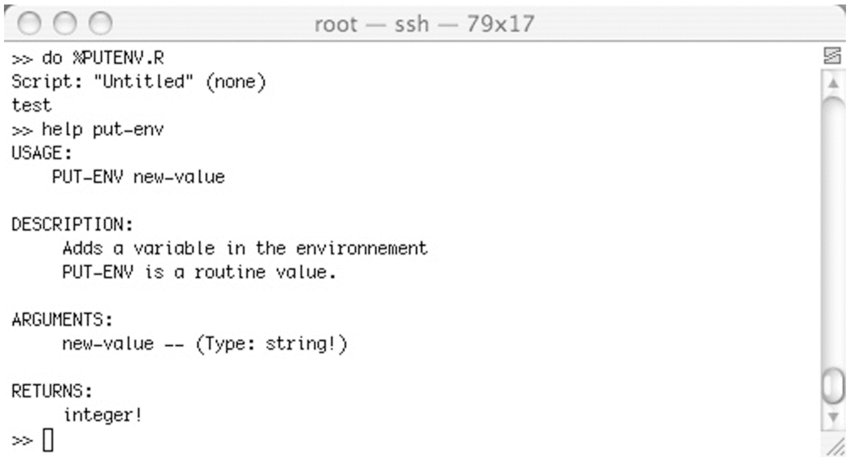
Providing parameters for a native function does not pose any great difficulty as long as they comply with the mapping of C and Rebol datatypes.

C Language	Rebol
Char	Char !
Short	Integer !
Long	Integer!
Int	Integer!
Float	Decimal!
Double	Decimal!
Struct*	Struct!
Char*	String!

Rebol and C equivalent datatypes

The following example declares the `put-env` word which adds a variable to the execution environment. Note also that the word is documented to be interrogated using the `help` word:

```
lib-so: load/library %libc.so.6

put-env: make routine! [
      “Adds a variable in the environnement"
      new-value [ string! ]
      return: [ integer! ]
] lib-so "putenv"

put-env "MYVARIABLE=test"
print get-env "MYVARIABLE"

free lib-so
```



```
>> do %PUTENV.R
Script: "Untitled" (none)
test
>> help put-env
USAGE:
    PUT-ENV new-value

DESCRIPTION:
     Adds a variable in the environnement
     PUT-ENV is a routine value.

ARGUMENTS:
     new-value -- (Type: string!)

RETURNS:
     integer!
>> 
```

Figure 6-8. *The word put-env is added to the Rebol dictionary.*

Obviously, some declarations are much more complex than the preceding examples. Numerous functions use structures to receive or return data. For this reason, Rebol includes the `struct!` datatype. If we want to use the `getpwuid` function from `libc` to find out information about the user identified by his UID, we must declare a structure to receive the returned data.:

```
lib-so: load/library %libc.so.6

get-uid: make routine! [
        return: [ integer! ]
] lib-so "getuid"

getpwuid: make routine! [
        uid [ integer! ]
        return: [ struct! [
                pw_name [ string! ]
                pw_passwd [ string! ]
                pw_uid [ integer! ]
                pw_gid [ integer! ]
                pw_gecos [ string! ]
                pw_dir [ string! ]
                pw_shell [ string! ]
        ] ]
] lib-so "getpwuid"

my-uid: get-uid
print [ "My UID is: " my-uid ]
me: getpwuid my-uid
print [ "My Login is:" me/pw_name ]

free lib-so
```

This section has demonstrated the ease with which Rebol is able to exploit the speed of a specific platform and its excellent integration with the Unix environment. With minimum code, Rebol permits the easy development of light, powerful applications which can be thoroughly integrated into their operating environment. The language's versatility makes it a tool of choice on all Unix systems.

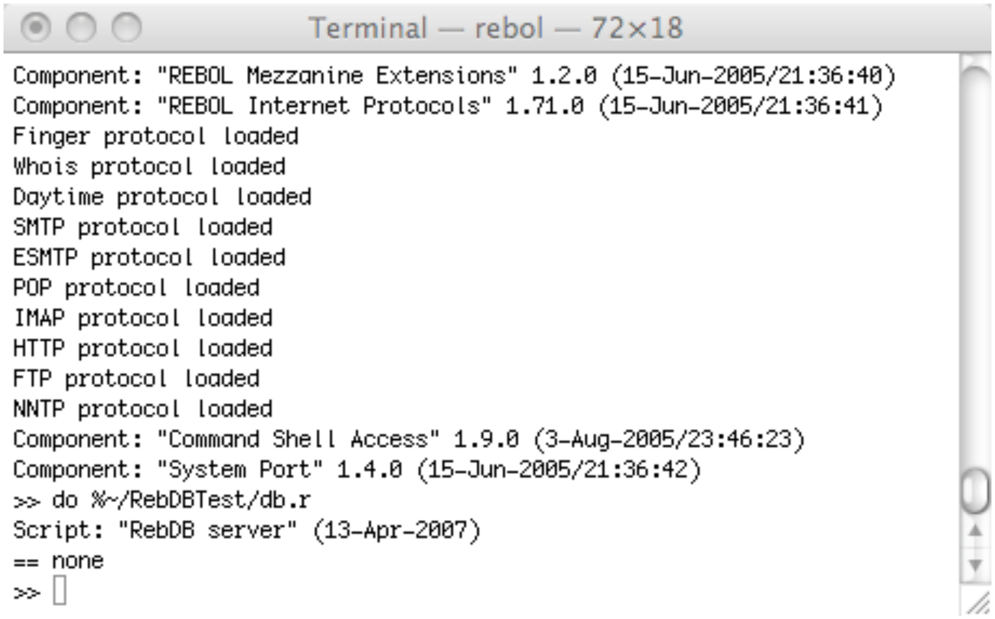
Databases with RebDB

RebDB was, and is still being, developed by Ashley Trüter. It's a database engine that can be directly integrated into a Rebol application and also used in the client/server model via TCP/IP. Lightweight and free, RebDB is even free to use when the end product is of a commercial nature. You can download it from Ashley's website http://www.dobeash.com/RebDB/ in the form of a simple zip archive. The project is carefully documented since a database guide and an SQL guide are available on the author's site. The documents quickly show that RebDB is not a toy but a remarkable product with a wealth of features available to developers. Compatible with all Rebol versions, it offers the ability to create data tables, manipulate information using SQL-like syntax and even distribute data by working in client-server mode. With RebDB, any Rebol application can therefore embed a real database engine that weighs only seventy kb. Written entirely in Rebol, this engine is highly optimised by building on the language's strong points. It also takes advantage of the language's portability and can operate in the same way on any of the platforms supported by Rebol.

Getting started with RebDB

RebDB consists of three Rebol scripts; db.r is the main database engine, db-client.r is the client module when implementing in client/server mode and SQL.r which provides SQL console access to RebDB databases. By default, RebDB stores databases in the directory from which the engine was loaded. This can be changed but in order to take advantage of RebDB's automated recovery features it is best to stick with the default behaviour. Therefore it is generally best to create a directory which includes a copy of all the RebDB scripts for each of your databases.

Once you've done that, all that is needed to load the database engine is `do %filepath/db.r` where *`filepath`* is the file path of the database directory relative to the current directory.



```
Component: "REBOL Mezzanine Extensions" 1.2.0 (15-Jun-2005/21:36:40)
Component: "REBOL Internet Protocols" 1.71.0 (15-Jun-2005/21:36:41)
Finger protocol loaded
Whois protocol loaded
Daytime protocol loaded
SMTP protocol loaded
ESMTP protocol loaded
POP protocol loaded
IMAP protocol loaded
HTTP protocol loaded
FTP protocol loaded
NNTP protocol loaded
Component: "Command Shell Access" 1.9.0 (3-Aug-2005/23:46:23)
Component: "System Port" 1.4.0 (15-Jun-2005/21:36:42)
>> do %~/RebDBTest/db.r
Script: "RebDB server" (13-Apr-2007)
== none
>>
```

Figure 6-9. *Loading the RebDB protocol.*

Once this simple operation is completed, a set of new words, prefixed with the characters "db-", is added to the dictionary of your interpreter. You can list these words by entering the command `help db-` in the Rebol console. The most effective way to learn about RebDB is to work directly in the console to see how each of the functions performs. The results of each data request or maintenance command are redirected to the console to be displayed.

Tables and fields

To organise and store data, RebDB uses the traditional model of tables. In these tables each column is a field and each row is a record (composed of one or more fields). The number of tables that can be created is limited only by the memory capacity of your machine. In effect, although RebDB uses files for data backup, it stores all the information in memory in order to minimise processing time. RebDB is therefore very effective for databases with small records.

On the other hand, if your databases are made up of many thousands of gigabytes, it would be better to turn to tested and tried alternatives such as MySQL or PostgreSQL.

One aspect of RebDB that gives it the speed to handle many rows of data is that when a table changes it doesn't automatically save the table to disk. Instead, it writes the changes to a log file and leaves it up to the programmer to save tables to disk by using the command `db-commit` followed by the table name. If your program finishes without saving tables changes, the database can be quickly recovered by using the command `db-replay` followed by the name of the table. If the database is held in the same directory as the `db.r` script, RebDB automatically issues a `db-replay` command on start-up to recover any uncommitted changes.

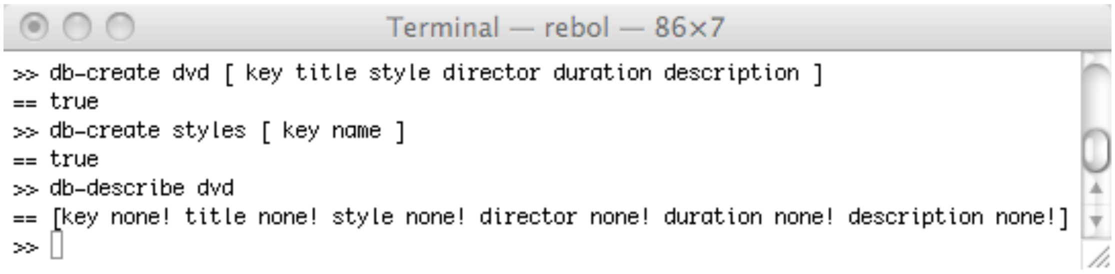
If, at any time, you want to undo any changes to the database that you haven't yet committed, you can do so easily with the help of the `db-rollback` command which takes a table name as its parameter.

All you need to do to create a table is use the word `db-create` followed by the name of the table and a list of its columns. You don't have to bother specifying the field type or length. In Rebol, values not variables have types. RebDB is built to take full advantage of this. Neither is there a need to specify keys or indexes with RebDB. Its optimised, in-memory approach makes them redundant.

Why not use this introduction to RebDB to set up a small DVD management application? Start by creating two tables that will be useful, that is a `dvd` table and a `styles` nomenclature table.

```
db-create dvd [ key title style director duration description ]
db-create styles [ key name ]
```

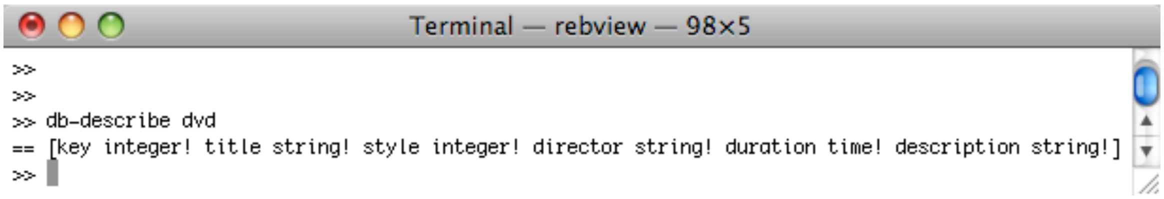
Once these two table have been created, you should see some new files in your current directory. The files with the extension .dat contain data, whilst those with the .ctl extension contain the structure of the tables. For each table, you must have both a .ctl file and a .dat file. You can check the structure of a table by using `db-describe`. It provides a Rebol block showing the name and type of each column.



```
>> db-create dvd [ key title style director duration description ]
== true
>> db-create styles [ key name ]
== true
>> db-describe dvd
== [key none! title none! style none! director none! duration none! description none!]
>>
```

Figure 6-10. *Checking the structure of a table.*

You will have noticed that the types of all the columns in the table have been set to `none!`. Column types are more informational than mandatory as in standard SQL databases. They reflect the types of the values in the first row of the table as we'll see below. In fact, RebDB will happily let you mix types in a column. Though if you do, don't be surprised when RebDB's built-in aggregating functions like `sum` and `avg` give strange results.



```
>>
>>
>> db-describe dvd
== [key integer! title string! style integer! director string! duration time! description string!]
>>
```

Figure 6-11. *The structure of a table once a row has been added.*

With `db-close`, the table indicated is unloaded from memory. RebDB includes a safety mechanism that only allows you to close tables that have not been changed since they last saved.

The word `db-drop` completely erases a table. Beware, the effects of this word are irreversible and your data will be permanently lost.

Manipulating data

Manipulating data in tables is entrusted to a set of words designated to inserting, modifying, selecting or deleting information. When not working in client-server mode, RebDB doesn't allow the user to write queries using SQL. But rest assured, the syntax chosen for the various words remain very close to SQL in both spirit and form. It takes only a few minutes to get comfortable with these new words.

Thus `db-insert` allows the insertion of a new row in a table. This word takes two arguments which are the name of the table and a block containing the values that are to be inserted into it. With `db-update`, you can change one or more fields in a given table. The `/where` *refinement* allows you to determine which records will be modified according to a Rebol expression which must evaluate to a Boolean value. Deleting data is entrusted to `db-delete` which works on a given table and also applies a Rebol Boolean expression in order to determine which data is to be erased. Finally, `db-select` searches for information in a table. Its two arguments are the names of the selected columns and the name of the table to be searched. This words allows the use of the word * to select all the fields in a table. Many refinements are available such as `/where` to make a conditional selection or `/order` and `/desc` to sort the data. The result is a block containing the data in each file for different rows.

Putting it to work

To better understand how RebDB works, you are going to build an application to manage a DVD collection with the help of Rebol/View. You have already created the `dvd` and `styles` tables used by your product. The script `MyDVD` begins by loading RebDB and verifying the presence of the database. In cases where the database does not exist, this code will build the database and initialise the styles nomenclature using predefined values. Inside the argument to the `db-insert` command, the use of the word `'next` indicates the value of the field is to be automatically incremented for each new record.

```
do %~/RebDBTest/db.r

if not exists? %dvd.dat [
 if error? try [
  db-create dvd [key title style director duration description]
  db-create styles [ key name ]
  foreach styledvd [
   "Action" "Adventure" "Humour"
   "Music" "Kids" ] [
     db-insert styles compose [ next (styledvd) ]
   ]
   db-commit styles
  ] [
   alert "Unable to create the database"
   quit
  ]
]
```

Your application should allow many operations. Four arrow buttons allow navigation through the database. The user can go directly to the first or last record and also go forwards and backwards one record at a time. To simulate a slider, the block `collection` is initialised by word `init-slider`.

```
init-slider: func [ /refresh /filter styl ] [
 either not filter [
  collection: db-select rowid dvd
 ] [ collection: db-select/where rowid dvd compose
      [ style = (styl) ] ]
 if not refresh [
  either (length? collection) > 0 [
   slider: 1
   modif: true
   aff-dvd
  ] [
   modif: false ; by default, we have inserted a new record
   slider: none
  ]
 ]
]
```

Given that the user can filter films by style, this word uses the `/filter` *refinement* which provides the appropriate selection of the DVDs. The `collection` block contains the line number of each record (`rowid`). When the user clicks on one of the arrow buttons, the `slider` variable is updated.. The various fields are displayed using `aff-dvd`. It retrieves the data by using `slider` as an index to the collections `block` and then retrieving the record with that `rowid.`

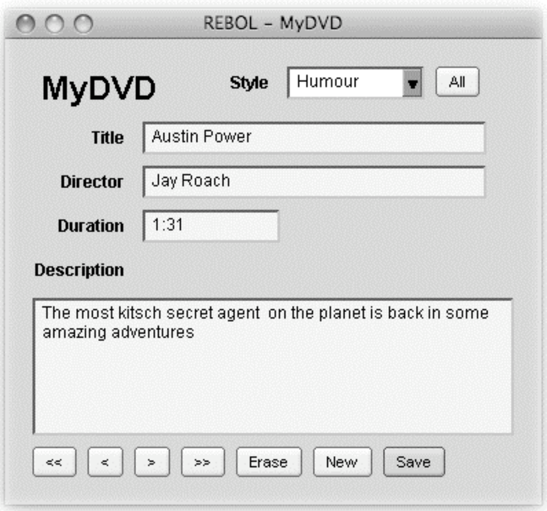


Figure 6-12. *The graphic interface of the"MyDVD" application.*

Another database query retrieves the style of the DVD by interrogating the nomenclature table. The updating of each field is performed using a innovation introduced in View version 1.3: "accessors'. These are the words `set-face`, `get-face` and `clear-face` which directly access the value of a graphical component.

```
aff-dvd: func [ /local data ] [
 data: db-select/where * dvd compose [
      rowid = (pick collection slider)
 ]
 set-face style-dvd db-select/where name styles compose [
      cle = (data/3)
 ]
 set-face title data/2
 set-face director data/4
 set-face duration data/5
 set-face description data/6
 modif: true
]
```

Editing and saving

Inserting or modifying a record takes place within the `layout` defining the graphical interface of your application. The "Save" button performs a `db-update` or a `db-insert` depending on the value of the boolean variable `modif`. In both cases, the program conservatively issues a `db-commit` to save the data to disk.

It must also search the `styles` table to find the key value of the style selected by the user in order to properly update the `style` field in the `dvd` table. The string-based search requires the use of a simple Rebol expression.

```
btn-enter "Save" [
 if error? try [
  either modif [
   db-update/where dvd [
    title style director
    duration description
   ] compose [
    (get-face title)
    (first db-select/where key styles compose [ name = (get-face style-
dvd) ] )
    (get-face director) (to-time get-face duration)
    (get-face description)
   ] compose [ rowid = (pick collection slider) ]
  ] [
   db-insert dvd reduce [
    'next (get-face title)
    (first db-select/where key styles compose [ name = (get-face style-
dvd) ] )
    (get-face director)
    (to-time get-face duration) (get-face description)
   ]
   init-slider/refresh
   slider: length? collection
  ]
 ] [ alert "Unable to save" ]
]
```

Deleting a record requires use of `db-delete`. The `/where` *refinement* allows you to select which line is to be deleted. The value of `key` is found by selecting the active element in the `collection` block. Depending on the case, the software displays either the record before the one that was deleted (or the new first record if it was the first record that was deleted) or empty fields. In the latter case, they are two possibilities to manage. In effect, it is possible that the database no longer contains any records and also that no records match the style chosen by the user.

```
btn "Erase" [
 if not none? slider [
  db-delete/where dvd compose [
   key = (first db-select/where key dvd compose [
    rowid = (pick collection slider)
   ] )
  ]
  db-commit dvd
  init-slider/refresh
  either (length? collection) > 0 [
   slider: slider - 1
   if slider = 0 [ slider: 1 ]
   aff-dvd
  ] [
   slider: none
   empty-fields
  ]
 ]
]
```

Publishing your database on the web

Why not share your DVD collection with friends and dynamically display your collection's different titles on your website? RebDB has a client-server mode that makes this very easy. This allows for simultaneous connections to the database, avoids loading tables for each client connected and has a SQL dialect that is quite close to the original. This latter includes the main commands SQL (`select`, `insert`, `delete` and `update`). It is thus possible to write complex queries such as:

```
select [ title director ]
from dvd
where [ duration < 2:00 ]
order by title desc
```

Installation takes only the creation of a `rebdb` directory and copying the RebDB libraries and your database files. Then you must write a short script to launch the RebDB server using the word `listen`. You are free to choose which TCP port to use.

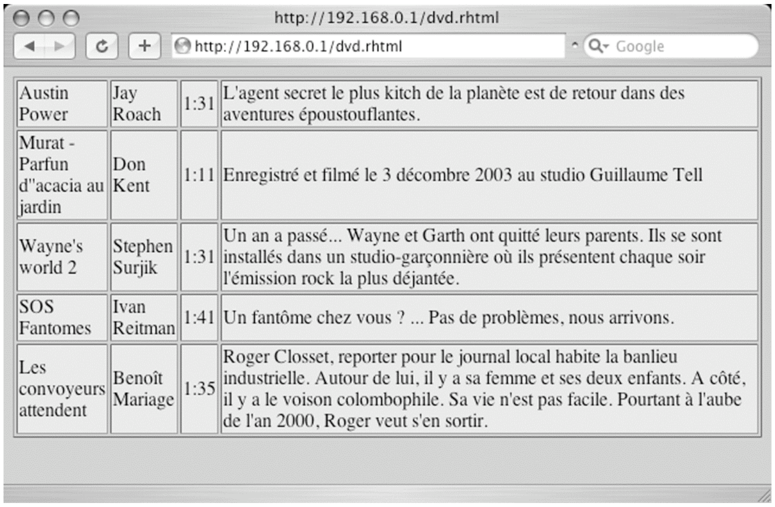
```
REBOL [ subject: "Launch RebDB server" ]
do %db.r
listen tcp://:10000
```

Once this script is launched, the RebDB server waits for connections from clients. Now all that is left is to write the dynamic page. To make you life easy, you will use the Magic! library. First, copy the RebDB files to the Magic! directory dedicated to data storage and sharing libraries. The `dvd.rhtml` page requires the RebDB client library to be available as well as the Magic! HTML components in order to present the data easily.

The request to the database is issued using the word `db-request` followed by a URL and the request itself.

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="magic.css">
</head>
<body>
<rebol>
 library %db-client.r
 library %html.r
 data: db-request tcp://192.168.0.1:10000 [
  select [ title director duration description ]  from  dvd
 ]
 block: copy []
 forskip data 4 [
      append/only block reduce [ data/1 data/2 data/3 data/4 ]
 ]
 html/datagrid block
</rebol>
</body>
</html>
```

The result of the request is assigned to the variable `data` and is then reformatted to match the data structure expected by the `datagrid` HTML component of Magic!.



Austin Power	Jay Roach	1:31	L'agent secret le plus kitch de la planète est de retour dans des aventures époustouflantes.
Murat - Parfun d''acacia au jardin	Don Kent	1:11	Enregistré et filmé le 3 décombre 2003 au studio Guillaume Tell
Wayne's world 2	Stephen Surjik	1:31	Un an a passé... Wayne et Garth ont quitté leurs parents. Ils se sont installés dans un studio-garçonnière où ils présentent chaque soir l'émission rock la plus déjantée.
SOS Fantomes	Ivan Reitman	1:41	Un fantôme chez vous ? ... Pas de problèmes, nous arrivons.
Les convoyeurs attendent	Benoît Mariage	1:35	Roger Closset, reporter pour le journal local habite la banlieu industrielle. Autour de lui, il y a sa femme et ses deux enfants. A côté, il y a le voison colombophile. Sa vie n'est pas facile. Pourtant à l'aube de l'an 2000, Roger veut s'en sortir.

Figure 6-13. *MyDVD display in a browser.*

RebDB is an effective solution for managing small Rebol databases and, most importantly, it helps to avoid the use of efficient but heavyweight, multidimensional products such as MySQL or PostgreSQL in small projects.

Not needing any configuration on the local machine, platform independent, compatible with all Rebol versions, useable in local or client/server mode, it simplifies the work of the programmer when data handling and storage becomes complex.

Summary

Rebol is full of original technologies. It's virtual offices can distribute applications and information over the Internet. In the Unix programming domain, Rebol has numerous interfaces to exploit the features and power of these operating systems. With RebDB, it is possible to develop *n-tier* applications written entirely in Rebol.

7
Practical applications

This chapter consists of a number of workshops to allow you to apply the knowledge you've acquired by now. These case studies focus on the foundations to build a video game, on developing a *chat* program, on writing a MySQL administration console, and finally on creating a reblet for Rebol/IOS.

Writing a raycasting engine with View

Rebol is a general language. You can write practically any program in Rebol. It is lightweight, handles data very effectively and possesses undeniable qualities in the field of network programming. It is also a gifted language in the graphics and animation fields. The GCS (*Graphical Compositing System)* is a truly platform-independent multimedia library. To illustrate its capabilities, you are going to make a real raycasting engine in only 3 kb of code.

What is raycasting?

Put simply, raycasting is one of the most intriguing aspects in the field of video games. Your memory may help you better understand... Some of you will remember a game called Wolfstein, that emerged in the 1990s. It was the first game where the player was immersed within its environment. Your character moves through a maze with the animation of the walls and NPC (non-player characters) done in real time.

Given the performance of machines of that time (Intel 286 and 386), it was simply incredible! How did John Carmack, the game's author, perform this miracle? Well the answer is simple: by cheating.

It looks real to me!

In fact, raycasting is a huge and great deception because it doesn't use a 3D algorithm at all. The player moves around a 2D universe whose screen visualisation is produced by launching rays.

The player is just moving around a two-dimensional table where each cell can be empty or represent a virtual cube defined by colour or texture. You simply define a field of vision whose width is usually set at 90 degrees, matching the minimum capability of a human being. Once this cone is created, you launch 90 rays to find the first intersection with a wall. All that is left is to determine the height of the wall segment depending on the distance and to draw it on the screen.

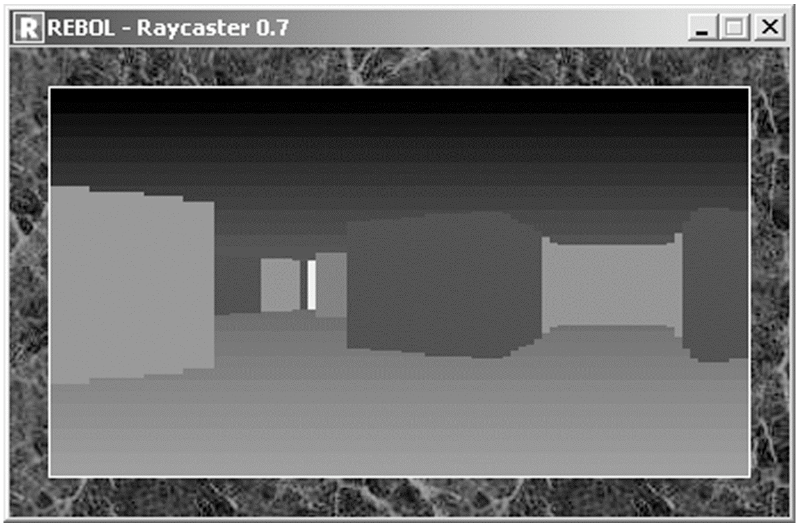


Figure 7-1. *The inside view of the labyrinth.*

In Wolfstein (and also our engine), the walls are designed to be symmetrical to the x-axis which represents the horizon. This algorithm has the advantage of simplicity but does not allow the player to look up or down.

Starting with the foundations

Before anything else, you must define the surface of the game and a number of variables and utility functions. The labyrinth (`laby`) is defined by a list of blocks (each block corresponds to a line of the table). If the value of a box is non-zero, this means that the box contains a cube.

To find the colour of this cube's walls, just look in the list initialised with the 16 colours of the famous but ancient Qbasic (`palette`). You also need to fix the player position (`px` and `py`), the distance covered in one step (`stride`), the direction of travel in degrees (`heading`) and the number of degrees when turning (`turn`). To speed up calculations, you also define a list containing the cosine values (`ctable`) for a certain number of angles. The function `get-angle` allows the extraction of a value from this table.

```
REBOL [
      subject: "raycasting engine"
      version: 0.7
]
px: 9 * 1024
py: 11 * 1024
stride: 5
heading: 0
turn: 10
laby: [
      [  8  7  8  7  8  7  8  7  8  7  8  7 ]
      [  7  0  0  0  0  0  0  0 13  0  0  8 ]
      [  8  0  0  0  12 0  0  0 14  0  9  7 ]
      [  7  0  0  0  12 0  4  0 13  0  0  8 ]
      [  8  0  4  11 11 0  3  0  0  0  0  7 ]
      [  7  0  3  0  12 3  4  3  4  3  0  8 ]
      [  8  0  4  0  0  0  3  0  3  0  0  7 ]
      [  7  0  3  0  0  0  4  0  4  0  9  8 ]
      [  8  0  4  0  0  0  0  0  0  0  0  7 ]
      [  7  0  5  6  5  6  0  0  0  0  0  8 ]
      [  8  0  0  0  0  0  0  0  0  0  0  7 ]
      [  8  7  8  7  8  7  8  7  8  7  8  7 ]
]

ctable: []
for a 0 (359 + 180) 1 [
      append ctable to-integer (((cosine a ) * 1024) / 10)
]

palette: [
      0.0.128 0.128.0 0.128.128
      0.0.128 128.0.128 128.128.0 192.192.192
      128.128.128 0.0.255 0.255.0 255.255.0
      0.0.255 255.0.255 0.255.255 255.255.255
]

get-angle: func [ v ] [ pick ctable (v + 1) ]
```

Several of these values are scaled by multiplying by 1024. This allows a player to move on each turn rather than jumping from box to box.

Shaping the walls

To draw on the screen, you need a window containing a picture. Here this is a `layout` called `screen`. The image used for the surface design is called `display` and measures 360 by 200 pixels.

You can already deduce that each ray launched will have a width of 4 pixels (360 / 90). A `gradient` allows the superimposition of a gradient to represent the floor and ceiling of you labyrinth. The function `refresh-display` calls the raycasting engine and displays the contents of the area.

```
refresh-display: does [
      retrace
      show display
]
screen: layout [
      backtile %marbre.jpg
      display: box 360x200 effect [
            gradient 0x1 0.0.0 128.128.128
            draw []
      ] edge [
            size: 1x1
            color: 255.255.255
      ]
]
refresh-display
view/title screen join "Raycaster " system/script/header/version
```

The heart of the program lies in the `retrace` function. A `for` loop finds the 90 angles where a ray should be launched. A virtual line is drawn from the position of the player based on each angle in turn. At the first non-empty box (its value is non-zero), the function calculates the height of the wall segment and determines the position and dimensions of the rectangle which it represents. All that is left to do is to get the colour from `palette` and add drawing instructions in the `effect/draw` attribute of the image

```
retrace: does [
      clear display/effect/draw
      xy1: xy2: 0x0
      angle: remainder (heading - 44) 360
      if angle < 0 [ angle: angle + 360 ]
      for a angle (angle + 89) 1 [
            xx: px
            yy: py
            stepx: get-angle a + 90
            stepy: get-angle a
            l: 0
            until [
                  xx: xx - stepx
                  yy: yy - stepy
                  l: l + 1
                  column:  make integer! (xx / 1024)
                  line: make integer! (yy / 1024)
                  laby/:line/:column <> 0
            ]
            h: make integer! (900 / l)
            xy1/y: 100 - h
            xy2/y: 100 + h
            xy2/x: xy1/x + 3
            color: pick palette laby/:line/:column
            append display/effect/draw reduce [
                  'pen color
                  'fill-pen color
                  'box xy1 xy2
            ]
            xy1/x: xy2/x + 1
      ]
]
```

Finishing with interaction

You must now manage the keyboard so that you can move around your labyrinth. For this, you define a function, `evt-key`, which is inserted into the event handler. According to the key pressed (`up`, `down`, `left` and `right`), you call the function `player-move`. It validates the player's movement by checking whether the player collides with a wall and redraws the screen by calling the `refresh-display` function.

```
player-move: function [ /backwards ] [ mul ] [
      either backwards [ mul: -1 ] [ mul: 1 ]
      newpx: px - ((get-angle (heading + 90)) * stride * mul)
      newpy: py - ((get-angle heading) * stride * mul)
      c: make integer! (newpx / 1024)
      l: make integer! (newpy / 1024)
      if laby/:l/:c = 0 [
         px: newpx
         py: newpy
         refresh-display
      ]

]

evt-key: function [ f event ] [] [
       if (event/type = 'key) [
          switch event/key [
               up [ player-move ]
               down [ player-move/recule ]
               left [
                   heading: remainder (heading + (360 - turn)) 360
                   refresh-display
               ]
               right [
                   heading: remainder (heading + turn) 360
```

```
                    refresh-display
                ]
            ]
        ]
        event
]
insert-event-func :evt-key
```

The algorithm presented here is not the most effective but it is probably one of the simplest. There are still many opportunities for optimisation and, especially, the visual appearance can be greatly improved by the use of textures.

Program your “chat” in Rebol

Let’s continue exploring Rebol by developing a chat application that allows a group of people to have a real-time discussion on the Web. For this project, we will need to develop a client module with Rebol/View and a server module consisting of a CGI script written in Rebol/Core.

Basic principles

The concept is fairly simple: each client uses a utility written with Rebol/View to provide a graphic user interface. The user, identified by name (*login*), can enter a message and send it to a server. Periodically, the client module checks for new messages on the server, retrieves them and displays on the screen. Users are able to chat in realtime.

It is important to note the technique chosen is that in which the clients ask the server if new messages are available. At no time does the server contact the clients to let them know that messages are waiting. With this mode of operation, based on request/response, we can easily use CGI scripts and especially the HTTP protocol.

Our client can therefore make its requests on TCP port 80 enabling it to cross any *firewall* encountered unhindered. Our chat system is perfectly usable on the Internet and not just a private network.

Writing the client

The first stage consists of asking the user what is there name. This can be done using a simple dialog box and assigning the value entered to the variable `login`.

```
login: request-text/title "Your login:"
```

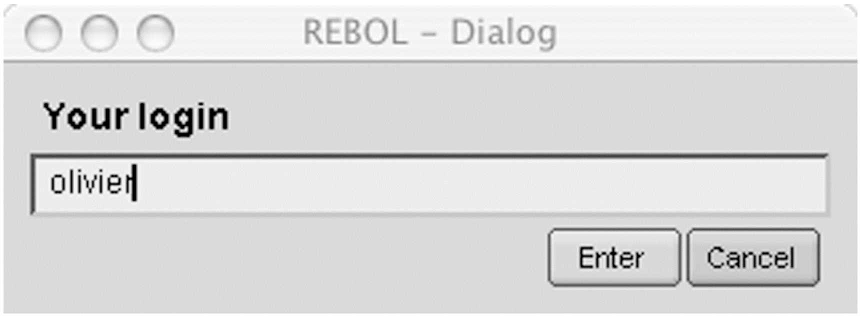


Figure 7-2. *Entering the user login.*

The main View client consists of a window containing a text-entry line, a button to send the entered text and a list showing messages with the sender's name. We therefore need to define a layout measuring 500 by 500 pixels containing three graphic components.

```
view layout/size [
      origin 0x0 space 0x0
      msgs: text-list 500x470 rate delay feel [
            engage: func [ f a e ] [
                  if a = 'time [ read-messages ]
            ]
      ]
      across
      txt: field 400x30
      button "Send" 100x30 [ send-message ]
] 500x500
```

A *timer* is defined within the component named `msgs`. The frequency of the timer being triggered is determined by the variable `delay` which contains the number of seconds separating two client connections to the server. The `engage` function then calls the `read-message` function if and only if the event is detected to be of the `'time` type.

For the button, we indicate that when the user clicks on it, the `send-message` function is called. this makes a request to the remote HTTP server using the POST method and passes it the message, the sender's name and a descriptive action indicating that the user is sending a new message. The server's URL is contained in the variable `server` defined at the start of the script.

```
server: http://jupiter/cgi-bin/rchat.cgi

send-message: does [
      read/custom server reduce [
            'POST
            join {action=SEND&login=} [
                  login "&message=" txt/text
            ]
      ]
      insert head msgs/lines join login [ ": " txt/text ]
      fix-slider msgs
      txt/text: copy ""
      show [ txt msgs ]
]
```

We need to properly initialise the message list slider. For this, a utility function called `fix-slider` adjusts the position and height of the slider on the vertical bar according to the number of messages received.

```
fix-slider: func [ faces ] [
      foreach list to-block faces [
            either 0 = length? list/data [
                  list/sld/redrag 1
            ] [ list/sld/redrag list/lc / length? list/data ]
      ]
]
```

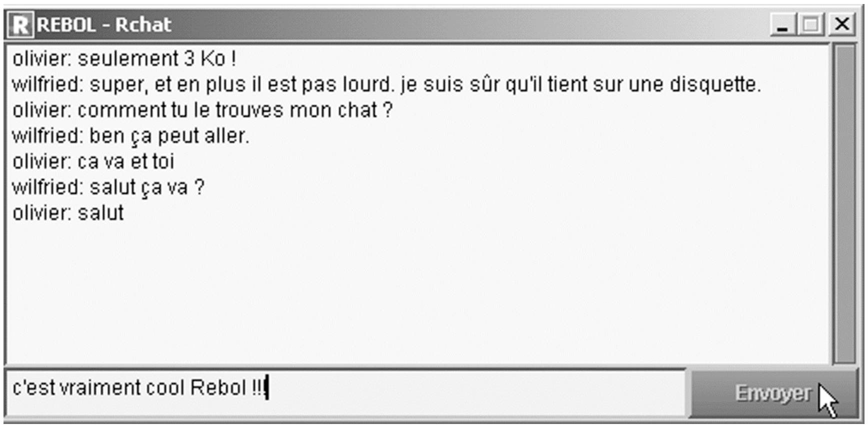
Once sent, the contents of the message line are erased and the message is added to the list. These two components are refreshed on the screen by using the word `show`.

All that remains is to write the `read-messages` function that retrieves messages sent by others users from the server.

This action is defined as "READ" and the user name is transmitted. The CGI script then returns all the messages written by other people connected to the server in a page with the MIME type of `text/plain`.

The variable `num-msg` contains the number of the last message read: the server only returns messages with a number greater than that in this variable.

```
read-messages: does [
      responses: read/custom server reduce [
            'POST
            join {action=READ&login=} [ login "&num=" num-msg ]
      ]
      if (length? trim responses) > 0 [
            resp: do responses
            num-msg: (first resp) + 1
            msg: second resp
            forskip msg 2 [
                  insert head msgs/lines join msg/1 [ ": " msg/2 ]
            ]
            fix-slider msgs
            show msgs
      ]
]
```


REBOL - Rchat

olivier: seulement 3 Ko !
wilfried: super, et en plus il est pas lourd. je suis sûr qu'il tient sur une disquette.
olivier: comment tu le trouves mon chat ?
wilfried: ben ça peut aller.
olivier: ca va et toi
wilfried: salut ça va ?
olivier: salut

c'est vraiment cool Rebol !!!

Figure 7-3. *Rchat in use.*

When the script starts to run, the variable `num-msg` is initialised by making a request with the action "NUM" to ascertain the number of the last message on the server.

```
num-msg: trim/all read/custom server [ POST "action=NUM" ]
num-msg: (to-integer num-msg) + 1
```

Setting up the server

On the HTTP server, we are going to write a CGI script called `rchat.cgi` and save it in the `cgi-bin` directory from where it will be run.

This script generates a page with a mime type of `text/plain` to return to the client. Its first job is to read the information received using the POST method and then extract the data using the `decode-cgi` word.

```
#!/usr/bin/rebol -cs
print "Content-Type: text/plain^/"

data: copy ""
len: to-integer system/options/cgi/content-length
until [
      buffer: copy ""
      read-io system/ports/input buffer (to-integer
      system/options/cgi/content-length)
      append data buffer
      ((length? data) = len)
]

query: make object! decode-cgi data
```

To permanently store data, the CGI uses two files called `num.txt` and `msg.txt`. The first contains the number of the last message received. The second stores messages in the format `number [ login message ]`. To avoid problems, the CGI script initially checks that the file msg.txt exists by trying to read it. If it encounters a problem, these two files are created.

```
if error? try [
      read %msg.txt
] [
      save %msg.txt [ ]
      write %num.txt 0
]
```

All that remains is to respond to the user action using a `switch` structure. If `query/action` contains “NUM”, the CGI script returns the value contained in the `num.txt` file.

In the case “SEND”, the CGI script formats the message and saves it in the `msg.txt` file and increments the value stored in `num.txt`. Finally, if the action is “READ”, the CGI script generates a block containing the number of the last message so that the client can update its `num-msg` variable and a series of blocks, each corresponding to a message. Thanks to *login*, the script only returns messages sent by other users to the client.

```
switch query/action [
      "NUM" [
            print read %num.txt
      ]
      "SEND" [
            num-msg: (make integer! read %num.txt) + 1
            write %num.txt num-msg
            bloc: load %msg.txt
            append bloc num-msg
            append/only bloc reduce [ query/login query/message ]
            save %msg.txt bloc
      ]
      "READ" [
            if error? try [
                  msglist: find (load %msg.txt) (do query/num)
                  msg: [ ]
                  forskip msglist 2 [
                        if (make string! msglist/2/1) <>
                  query/login [
                              append msg msglist/2
                        ]
                  ]
                  either all [ (not none? msg) ((length? msg) > 0) ] [
                        print mold reduce [ (to-integer trim/all
                  read %num.txt) msg ]
                  ] [ print "" ]
            ] [ ]
      ]
]
```

We have developed, in under 4 kb of code, a full operational chat client and server which are capable of operating directly over the Internet. This code can be the base for many other projects. You can improve it to manage concurrent access to data files and add features (emoticons, differently colour messages in the list, ensuring no two identical names, etc.). It can also be used as a base for online games written in Rebol.

In fact, interfacing View with CGI scripts is so simple and mostly intuitive that you can use this approach in many situations and build real applications using a distributed architecture.

A MySQL administrative console

In an earlier chapter, we explored using the core functionality of Nenad Rakocevic's excellent MySQL library.

As a reminder, it allows the free versions of Rebol (Core and View) to execute SQL queries against a MySQL database. We will now implement more advanced functions, for remotely administering a remote database, to build a small management console with Rebol/View.

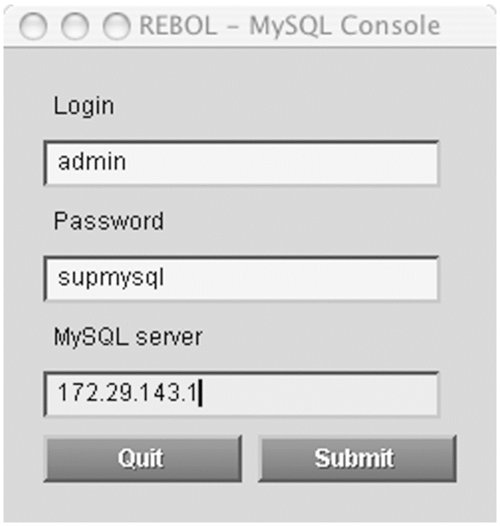
The specification

Our administrative console is a tool that allows us to connect to a MySQL server in order to visualise its databases. For each of these, we must be able to understand the tables and run SQL queries. Like any good administrator, we also want to collect DBMS usage statistics, be able to stop the DBMS and finally, to verify the connection between MySQL and the client. We will meet these ambitious specifications in less than 3 kb of code.

User identification

To secure the data it is hosting, MySQL identifies the rights of users with accounts and passwords. Our administrative console must therefore begin its work by identifying a user and selecting a MySQL server. This information is stored in the `text` properties of the `login`, `password` and `server` words. Once they have been entered, the “Submit” button retrieves a list of the databases available on the server and displays the main console window.

```
view/title layout [
     text "Login"
     login: field ""
     text "Password"
     password: field ""
     text "MySQL server"
     server: field ""
     across
     button "Quit" [ quit ]
     button "Submit" [
            load-bases
            view/title main "MySQL Console"
     ]
] "MySQL Console"
```



Login: admin
Password: supmysql
MySQL server: 172.29.143.1

Figure 7-4. *User identification*

Creating the main screen

Our administrative console consists of a single window called `main`. It has three buttons whose functions are to send commands to MySQL (statistics, connection status, and stopping the DBMS). A fourth button allows the user to quit the application.

Two lists (`bases` and `tables`), positioned with two tabs by the `tabs` instruction, show the databases managed by the MySQL server to which we are connected and the selected database. An input field, `reqsql`, allows an SQL command to be keyed in and executed by clicking on the provided "Enter" button.

```
main: layout/size [
    across
    button "Statistics" [ send-command 'statistics ]
    button "Connection ?" [ send-command 'ping ]
    button "Stop MySQL" [ send-command 'shutdown ]
    button "Quit" [ quit ]
    return
    tabs [ 10 250 ]
    text "Bases"
    tab text "Tables" return
    bases: text-list 220x100 [
           load-tables bases/picked
    ]
    tab tables: text-list 220x100 [
        reqsql/text: join "select * from " (first tables/picked)
        show reqsql
    ]
    return
    text "SQL request:"
    across
    reqsql: field 250 ""
    button "Enter" [ user-request ]
] 500x230
```

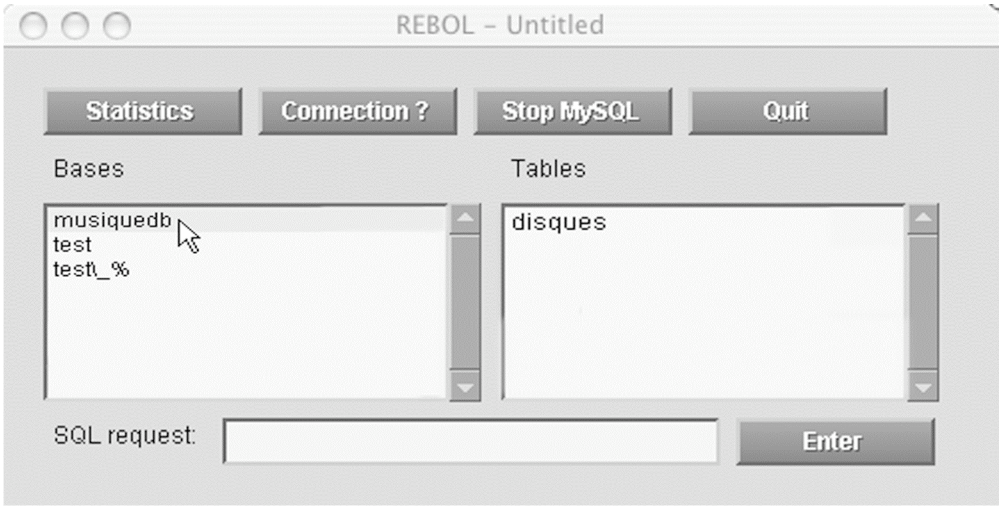


Figure 7-5. *The musicdb database contains the discs table.*

Some utility functions

To make things easier, we are going to define three utility functions. First, we need to correctly position the slider of the two lists of the `main` window when their contents are updated. For this, we must define a function, `fix-slider`, which takes the name of the list whose contents have changed as a parameter.

```
fix-slider: func [ faces ] [
    foreach list to-block faces [
            either 0 = length? list/data [
                    list/sld/redrag 1
            ] [
                    list/sld/redrag list/lc / length? list/data
            ]
    ]
]
```

Our application must frequently connect to the MySQL server. So we obviously need a function that automatically generates the URL connection from the user's account, password, server name and the MySQL database to be accessed. This function, `construct-url`, takes the name of the database as a parameter.

```
construct-url: func [ base ] [
      return join mysql:// [ login/text ":" password/text
            "@" server/text "/" base
      ]
]
```

Finally, it will be very helpful to have a function whose role is to run an SQL request. The `do-request` function takes in the SQL request and the name of the database to which it is to be forwarded. It connects to the database after calling `construct-url`. The data collected from the database is returned to the caller.

```
do-request: function [ base req ] [ p data ele ] [
        db: open construct-url base
        insert db req
        data: copy db
        close db
        return data
]
```

The processing functions

We will start by sending commands to the MySQL server. Nenad's library provides a set of features that support the management of a remote MySQL server. These commands are directly inserted into the communications port. They take the form of a block whose first element is the command. It is followed by any optional arguments. You can change the active database (`init-db`), switch user (`change-user`), create a new database (`create-db`), destroy a database (`drop-db`), update the MySQL server parameters (`reload`), end a process (`process-kill`) and put the server into "development' mode with the command `debug`.

For our console, we will use the `statistics`, `ping` and `shutdown` commands. These are passed as parameters, in the form of a symbol, to the `send-command` function.

It generates a block containing supplied options if the `/options` *refinement* is used. The result of the command is displayed in an alert box.

```
send-command: function [ cmd /options opts ] [ db bloc ] [
    if error? try [
       db: open construct-url "mysql"
       bloc: copy []
       append bloc reduce cmd
       if options [ append bloc reduce opts ]
       res: insert db bloc
       close db
       if not none? res [ alert to-string res ]
    ] [
       alert "It was not possible to carry out the command"
    ]
]
```

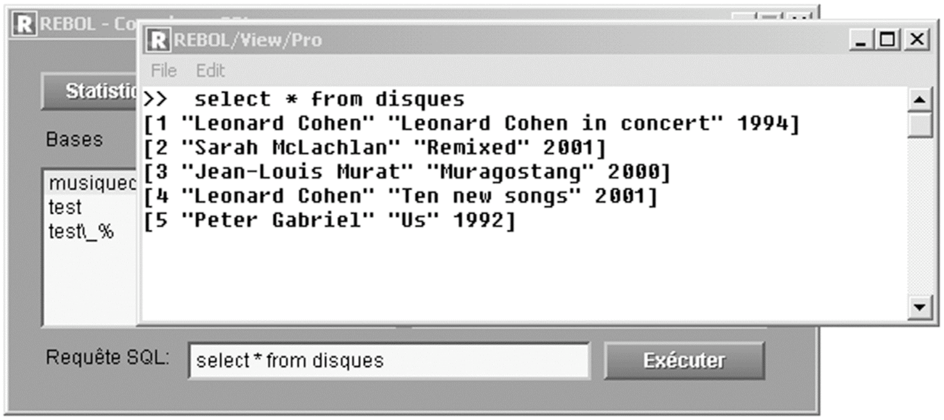
The `load-bases` and `load-tables` functions are used to fill out the `bases` and `tables` lists of the `main` layout. The names of the databases can be retrieved simply by making an SQL request against the `db` table of the MySQL database. As against retrieving the list of tables in a database which can only be done with the `show tables` command after `init-db` has been used to select the active database.

```
load-bases: does [
    clear bases/lines
    bases/lines: copy []
    foreach ele do-request "mysql" "select db from db" [
            append bases/lines ele
    ]
    fix-slider bases
    show bases
]

load-tables: func [ base ] [
    send-command/options 'init-db base
    tables/lines: copy []
    foreach ele do-request base "show tables" [
            append tables/lines ele
    ]
    fix-slider tables
    show tables
]
```

Executing an SQL request input by the user is done by the `user-request` function.

It retrieves the value of the `reqsql` input field and displays the lines read from the database. Using `mold` keeps the on-screen formatting of the data in the form of blocks.

```
user-request: does [
   either not empty? bases/picked [
          send-command/options 'init-db (first bases/picked)
          print [ "^/>> " reqsql/text ]
          foreach ele do-request (first bases/picked)
      reqsql/text [
                  print mold ele
          ]
   ] [ alert "You must select a database" ]
]
```



```
>>  select * from disques
[1 "Leonard Cohen" "Leonard Cohen in concert" 1994]
[2 "Sarah McLachlan" "Remixed" 2001]
[3 "Jean-Louis Murat" "Muragostang" 2000]
[4 "Leonard Cohen" "Ten new songs" 2001]
[5 "Peter Gabriel" "Us" 1992]
```

Figure 7-6. *The result of an SQL request is displayed in the console.*

In less than 3 kb of code, you have developed a visual program to drive a MySQL server.

As you've observed, building a genuine MySQL administrative tool is very easily achievable with Rebol.

Writing a reblet for IOS

The interesting principle behind IOS is that it is not static or frozen. With minimum investment, it is quite conceivable to write independent applications that utilise the capabilities of this application server. IOS provides a rich and powerful API to facilitate writing reblets.

The specification

Your first Rebol/IOS development project is a telephone directory which allows your IOS users to share information about their contacts. This information will not only be stored on the server but also synchronised with each of the Rebol/Link clients. A mobile user can recover the contacts added to the server since their last connection and can even view the information when if they are not connected to the IOS server. Your reblet will securely transport its data over networks because all information is automatically encrypted by IOS.

The development cycle

To write reblets, you only need a Link client and a proper code editor. On the other hand, your IOS user profile must allow you to publish new applications. This is done through the `new-app` option of your account.

First, create a directory on your hard disk to store your project code and resources needed.

Now, let us step back for a moment to consider what exactly a reblet is and how they work.

In fact, a reblet is simply a Rebol script downloaded over the network and executed on the client. IOS introduces the additional concept of communicating with the application server. When the reblet is executed in the environment established by Link, it may ask the IOS server to perform a number of operations such as reading or deleting a file. Reblets can also ask IOS to execute code on the server. This is done by a specific method called POST.

All such communication is performed using the asynchronous model allowing the client code to continue its execution without waiting for the response from the server.

To publish reblets, you only need to design an installation script which will create the environment to host them on the server (*fileset*), save the client code on the server and possibly create an IOS POST method. It is very important to remember that at no time, do you need to work on the server. All these operations can be performed using the Link client.

Defining a fileset

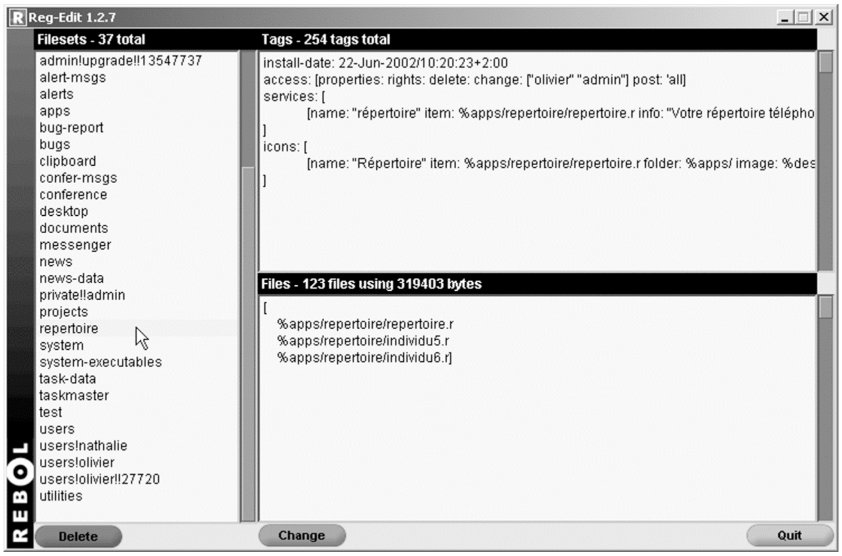
The first stage consists of writing a script called `install-phonebook.r` whose mission is to create a *fileset* called `phonebook` for our application and to install a POST method on the server. The REBOL header block of the script must contain the type `'link-app` to indicate to Link that the application uses IOS server features.

The *fileset* is sent to the server with the help of `install-fileset`. It gives editing rights only to users "olivier" and "admin". On the other hand, all users of the server are given the right to perform the POST method to able to publish their data.

The `icons` tag indicates the application name and icon which Link will display on the desktop. `files` contains the list of files which make up the client application.

```
fileset: 'phonebook
tags: [
	access: [
		properties: rights: delete: change: [
			"olivier" "admin"
		]
		post: 'all
	]
	icons: [
		[
			name: "Phonebook"
			item: %apps/phonebook/phonebook.r
			folder: %apps/
			image: %desktop/icons/demo.gif
			info: "Your phone book."
		]
	]
]
```

```
files: [ [%apps/phonebook/phonebook.r %phonebook.r] ]
```



Filesets - 37 total

```
admin!upgrade!!13547737
alert-msgs
alerts
apps
bug-report
bugs
clipboard
confer-msgs
conference
desktop
documents
messenger
news
news-data
private!!admin
projects
repertoire
system
system-executables
task-data
taskmaster
test
users
users!nathalie
users!olivier
users!olivier!!27720
utilities
```

Tags - 254 tags total

```
install-date: 22-Jun-2002/10:20:23+2:00
access: [properties: rights: delete: change: ["olivier" "admin"] post: 'all]
services: [
        [name: "répertoire" item: %apps/repertoire/repertoire.r info: "Votre répertoire télépho
]
icons: [
        [name: "Répertoire" item: %apps/repertoire/repertoire.r folder: %apps/ image: %des
]
```

Files - 123 files using 319403 bytes

```
[
  %apps/repertoire/repertoire.r
  %apps/repertoire/individu5.r
  %apps/repertoire/individu6.r]
```

Figure 7-7. *The fileset of the Phonebook application.*

The POST method

The block `post-func` contains the code of the POST method. The local variables used by the latter are given to the server using the `post-locals` list. It is better to create a separate context for this code to preserve the server's dictionary and protect the code from name clashes.

```
post-locals: [ base file num ]
post-func: [
      if error? err: try [
            server-code: context [
                  num: 0
                  base: %apps/phonebook/
                  if data-exists? base/num.r [
                        num: load-data base/num.r
                  ]
                  write-data base/num.r mold num + 1
                  file: join "individual" [ num ".r" ]
                  add-file 'phonebook base/:file (compress message/1)
                  true
            ]
      ] [
            print mold/only disarm :err
            false
      ]
]
```

The role of this piece of code is to receive data from the Link client and create a file on the *fileset* of the server. Each entry in the phonebook will be saved in a file `individualX.r` (x is a number incremented by 1 for each new entry). On the server, a file called `num.r` is generated by this method and contains the number of the last number used. You've probably noticed the presence of some new words that are part of the IOS server API:

- `data-exists?` checks for the presence of a data file,
- `write-data` creates a data file stored only on the server,
- `add-file` adds a file to an applications *fileset* (this new file will be automatically synchronised when clients connect to the server).

The client code

You can now proceed to write the client code to be stored in the file named `phonebook.r`. Once more, the Rebol header block must contain `'link-app` type to identify itself as an IOS reblet. The word `connected?` lets you find out if the client is connected to the server or not. If not, the publication functions are disabled and the user can only see data previously synchronised by Link.

The variable `user-name` contains the user's Link login. This information is one of the many properties of `user-prefs`.

The application's user interface consists of a simple window with a drop-down list called `cts` (the slider is put in place by the `fix-slider` function), three text input fields (`name`, `first-name` and `tel`) and three buttons, one to clear the form, one to add a new record and the last to quit the application. Link introduced some new styles to VID such as rounded buttons (`btn`).

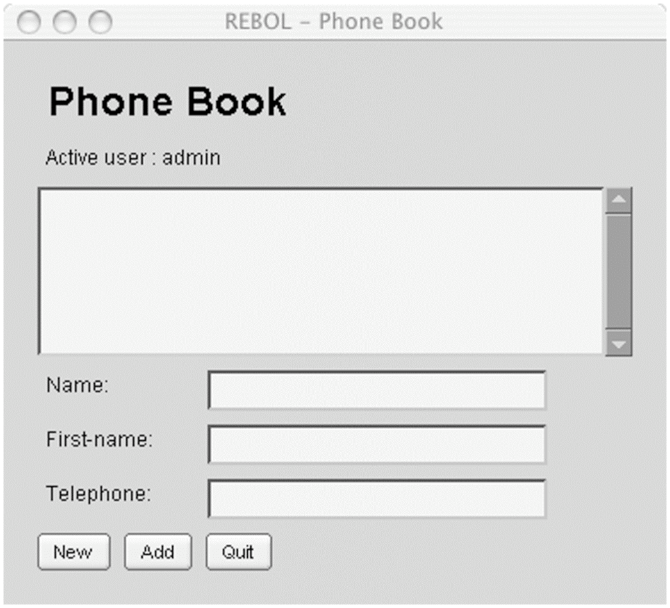


Figure 7-8. *The client portion of the Phonebook reblet.*

The use of the `new` *refinement* of `view` postpones the processing of events. Once the `layout` is displayed, the code continues its execution. This allows us to ask Link for the information it has recovered by connecting to the server. For this, we declare an event handler for the `get` command using the `insert-notify` word. The third parameter (`get-files`) corresponds to the name of a callback function invoked when the result of the request is received by the client. `get` is used to obtain a list of files in a given *fileset*. The `send-link` word sends the `get` command. Once the data are available, the `get-files` function is called It deactivates the event handler (`remove-notify`) and updates the list using the synchronised files that have been stored on the client.

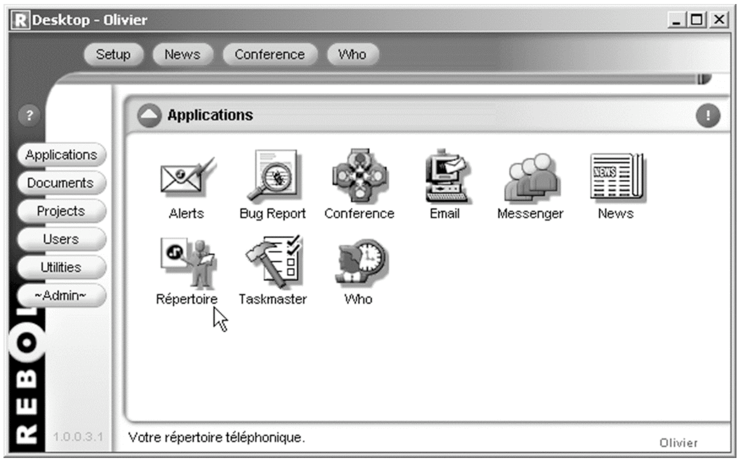
```
get-files: function [ event data ] [ list ] [
      remove-notify 'get 'phonebook
      list: copy []
      foreach individual data [
            if found? find individual "ind" [
                  append list (read join link-root individual)
            ]
      ]
      append cts/lines (sort list)
      show cts
]
insert-notify 'get none :get-files
send-link 'get 'app-files 'phonebook
```

Once this has been done, the processing of events which was previously postponed is started by with the `do-events` word.

A new record is created by calling the POST method on the serve by using the `send-server` command.

```
send-server post reduce [ 'phonebook new-cts ]
```

The arguments are the *fileset* to be used and the data to be transmitted. On the server, the POST method receives it using the `message` list.



Votre répertoire téléphonique.

Figure 7-9. *The Phonebook reblet is available on the Rebol/Link desktop..*

You've just completed writing your first reblet. All that is left to do is publish your work on the server. In Link, use the CTRL + L key combination and select the file called `install-phonebook.r`. The code is executed and your reblet is now available to your users.

Summary

Again, these themes have show the versatility of Rebol. You have developed a raycasting engine, written a perfectly functional chat client in a few lines of code, set up an MySQL manager and conceived a reblet for Rebol/IOS.