

Linux on IBM server zSeries and S/390: Application Development

Tools and techniques for Linux
application development

Using the Eclipse IDE and
Jakarta Project tools

Sample code to illustrate
programming techniques



Gregory Geiselhart
Andrea Grahn
Frans Handoko
Jörg Hundertmark
Albert Krzymowski
Eliuth Pomar



International Technical Support Organization

**Linux on IBM @server zSeries and S/390:
Application Development**

July 2002

Archived

Note: Before using this information and the product it supports, read the information in “Notices” on page xiii.

First Edition (July 2002)

This edition applies to zVM 4.2 (ESP) and many different Linux distributions. SuSE Linux Enterprise Server 7.0 was used.

© Copyright International Business Machines Corporation 2002. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xiii
Trademarks	xiv
Preface	xv
The team that wrote this redbook	xvi
Become a published author	xvii
Comments welcome	xvii
Part 1. Programming tools	1
Chapter 1. The basic tools you need	3
1.1 Where you can look for information	4
1.1.1 Man pages	4
1.1.2 Info - the help system	5
1.2 Compiling C/C++ source code	6
1.2.1 Starting gcc	6
1.2.2 Source files	6
1.2.3 Directory search	7
1.2.4 Compilation stages	7
1.2.5 Macros	8
1.2.6 Warnings	8
1.2.7 Extra information for debuggers	8
1.2.8 Code optimization	9
1.2.9 Configuring gcc as a cross-compiler	9
1.3 Linking object code	13
1.4 Automating the build process	14
1.4.1 GNU make	14
1.4.2 Writing your Makefile	14
1.4.3 Building with make	16
1.4.4 Makedepend	17
1.4.5 File dependencies from gcc	17
1.5 Libraries	18
1.6 Tracking changes	19
1.6.1 Using diff to find differences	19
1.6.2 Applying changes	21
1.6.3 Running diff against source tree	22
1.6.4 Distributing patches	24
1.6.5 Before you distribute your patch	25

Chapter 2. The IBM Java Software Development Kit	27
2.1 Java 2 Platform, Software Development Kit	28
2.1.1 References	28
2.2 IBM Java Developer Kit for Linux running on zSeries	28
2.2.1 Obtaining the IBM Java Developer Kit	28
2.2.2 Installing the IBM Java Developer Kit	28
2.3 Jikes	29
2.3.1 Installing Jikes	30
2.3.2 Using Jikes	30
Chapter 3. Source code control using CVS	33
3.1 Introduction to CVS	34
3.1.1 Definitions	34
3.1.2 Revision numbering	35
3.1.3 File locking	35
3.2 CVS command syntax	35
3.2.1 Global options	36
3.2.2 CVS commands	36
3.2.3 Command options	37
3.2.4 Command arguments	37
3.2.5 Log messages	38
3.2.6 Date formats	38
3.3 Administering CVS	38
3.3.1 Creating a repository	38
3.4 Root directory	39
3.5 ssh access	40
3.6 Environment variables and the ~/.cvsrc file	40
3.7 Creating a project	41
3.7.1 Importing the files	41
3.8 Obtaining a working copy	42
3.8.1 Special files	43
3.9 Making changes in the working copy	44
3.10 Adding files and directories	45
3.11 Committing changes to the repository	46
3.12 Updating the working copy	48
3.13 Resolving conflicts	51
3.14 Viewing log messages	51
3.15 Recovering versions	52
Chapter 4. The Emacs editor	55
4.1 Editing files using Emacs	56
4.1.1 Starting Emacs	56
4.1.2 Basic commands	57

4.1.3	Invoking Lisp functions	57
4.1.4	Editing multiple files	58
4.1.5	Moving text	59
4.1.6	Search and replace	61
4.1.7	Modes	61
4.2	Building applications using emacs	62
4.2.1	Editing program files	62
4.2.2	Compiling your application	63
Chapter 5. The vi editor		65
5.1	First encounter with vi	66
5.2	Modes of operation	66
5.3	Customizing vi	68
5.4	Commands categorized by functionality	68
5.4.1	Moving the cursor	68
5.4.2	Insertion point	69
5.4.3	Locating a string pattern	69
5.4.4	Replacing	69
5.4.5	Deleting	70
5.4.6	Moving and copying	71
5.4.7	Miscellaneous	71
5.4.8	Saving and closing file	71
5.5	To probe further.	72
5.6	An editor for the CMS aficionados.	72
Chapter 6. The Jakarta project.		75
6.1	The Tomcat application server	76
6.1.1	Obtaining Tomcat	76
6.1.2	Installing Tomcat	76
6.1.3	Configuring the Tomcat server	78
6.1.4	Deploying an application under Tomcat	79
6.1.5	Tomcat application manager.	79
6.2	Ant	80
6.2.1	Installing Ant	80
6.2.2	Using Ant.	80
6.3	Log4J	81
6.3.1	Installing Log4j	81
6.4	Taglibs	82
6.4.1	Installing taglibs.	82
6.4.2	Configuring taglibs	82
6.5	Struts	84
6.5.1	Struts components	84
6.5.2	Installing Struts	87

Chapter 7. Running Linux applications in a zSeries environment	89
7.1 Architecture consideration	90
7.1.1 Bits and bytes	90
7.1.2 Virtual address space	92
7.1.3 Function calling convention	93
7.2 When things go wrong	96
7.2.1 Debugging with gdb	96
7.2.2 Tracing system calls	101
7.2.3 Debugging under zVM	102
7.2.4 Performance profiling	102
7.3 Optimizing for performance	102
7.3.1 General options	102
7.3.2 Inline functions and unrolled loops	103
7.3.3 Architecture-dependent options	103
7.3.4 String operations	104
7.3.5 Sources of information	104
7.4 Signals	104
7.4.1 Linux signals and zSeries exceptions	105
Part 2. Eclipse	107
Chapter 8. Eclipse overview	109
8.1 Eclipse Software Developer Kit	110
8.2 The Eclipse platform	110
8.2.1 Ant	110
8.2.2 Compare	111
8.2.3 Core	111
8.2.4 Debug	111
8.2.5 Help	111
8.2.6 Release Engineering	111
8.2.7 Scripting	111
8.2.8 Search	112
8.2.9 Standard Widget Toolkit	112
8.2.10 User Interface	112
8.2.11 Update	112
8.2.12 Version Control Mechanism	112
8.2.13 WebDav	112
8.3 The Java Development Toolkit	113
8.3.1 JDT Core	113
8.3.2 JDT Debug	113
8.3.3 JDT UI	113
8.4 The Plug-in Development Environment	113
8.4.1 PDE Core	114

8.4.2 PDE UI	114
8.5 Getting started with Eclipse	114
Chapter 9. Installing Eclipse.	115
9.1 Prerequisite software for Eclipse	116
9.2 Eclipse installation	116
9.2.1 Rebuilding Eclipse	117
9.2.2 Build the Standard Widget Toolkit	118
9.3 Set up the environment	118
9.3.1 Testing the installation	119
9.4 Installing the C/C++ Development Tools plug-in	119
9.4.1 Installing the CDT client	121
9.4.2 Installing the CDT server code	121
Chapter 10. Configuring Eclipse	123
10.1 Starting Eclipse	124
10.1.1 The -vm option	124
10.1.2 The -data option	124
10.1.3 The -vmargs	124
10.1.4 Other start options	124
10.1.5 Simplifying options	125
10.2 Configuring Eclipse to use CVS	125
10.3 Eclipse and editors	128
10.4 Modifying Eclipse	128
10.4.1 Workbench	128
10.4.2 Perspectives and components	129
Chapter 11. Eclipse as an integrated development environment	131
11.1 Concepts	132
11.1.1 Workbench	132
11.1.2 Perspective	132
11.1.3 View	132
11.1.4 Editors	132
11.1.5 External editors	132
11.1.6 Resources	132
11.1.7 Graphical concept view	133
11.2 Using the Java Development Toolkit	134
11.2.1 Menu bar and tool bar	134
11.2.2 JDT initialization	135
11.2.3 JDT Java project	135
11.2.4 Running the application	139
11.2.5 Debugging the application	140
11.3 Using Eclipse with Ant	142
11.4 Using Eclipse with CVS	144

11.5 Using the C Development Toolkit	147
11.5.1 Sample project	147
11.5.2 Navigating code	149
11.5.3 Compiling the project	150
11.5.4 Running the code	150
11.5.5 Debugging the application	151
11.5.6 Packaging and managing projects	152
11.6 Using the Plugin Development Environment	153
11.6.1 Setting up the development environment	153
11.6.2 First plug-in	153
11.6.3 Making sense	154
11.6.4 Adding extensions	156
11.6.5 Running the plug-in	157
11.6.6 Deploying a plug-in	157
Part 3. Programming techniques	159
Chapter 12. zSeries as a development platform	161
12.1 Example applications	162
12.1.1 Application overview	162
12.1.2 The development environment	163
Chapter 13. Using the Struts framework	167
13.1 The Struts application components	168
13.2 The model component	169
13.2.1 User class	169
13.2.2 ActionForm class	170
13.2.3 Form validation and ActionErrors	171
13.2.4 Internationalization and application resources	171
13.3 The view component	172
13.3.1 Struts-html taglib	172
13.3.2 Mapping form input to ActionForm beans	174
13.4 The controller component	174
13.4.1 Action class	174
13.5 Logging using Log4j	176
13.5.1 Using Log4j	177
13.5.2 Configuring Log4j	177
13.6 Struts framework configuration	178
13.6.1 Registering ActionForm beans	178
13.6.2 Registering ActionMapping and ActionForward	179
13.6.3 Configuring ActionServlet	179
13.7 The persistence layer	180
13.7.1 Data abstraction in the persistence layer	181
13.8 The JDBC interface	182

13.9	Connection pooling	184
13.9.1	Connection configuration	186
13.10	The Java Native Interface	188
13.10.1	Using JNI in Java code	188
13.10.2	Implementing the native code in C	189
13.10.3	Building the JNI shared library	191
Chapter 14.	Shared libraries and more	193
14.1	Example overview	194
14.1.1	Components of the address book example	194
14.1.2	Implemented functionality	195
14.2	Creating and using libraries	196
14.2.1	Preparing object files	196
14.2.2	Inspecting object files	197
14.2.3	Static libraries	198
14.2.4	Shared libraries	199
14.2.5	Using shared libraries	200
14.2.6	Building shared libraries	202
14.2.7	Investigating shared object dependencies	202
14.2.8	Dynamically linked libraries	203
14.2.9	Include files	205
14.3	A poor man's database	207
14.3.1	Memory mapped files	207
14.3.2	Synchronizing memory and disk storage	210
14.4	Graphical user interface	211
14.4.1	Graphical interface in a UNIX environment	212
14.4.2	Qt library	213
Chapter 15.	Designing for concurrent access	217
15.1	UNIX processes	218
15.2	The pthreads library	219
15.2.1	Using threads	220
15.2.2	Creating threads	221
15.2.3	Thread termination	222
15.2.4	Thread attributes	224
15.2.5	Setting thread stack size	224
15.2.6	Synchronizing threads	226
15.2.7	Mutexes	227
15.2.8	Conditional variables	228
15.3	Controlling concurrent access	230
15.3.1	Locking using files	230
15.3.2	IPC semaphores	230
15.3.3	Pthread resources	231

Chapter 16. Concurrency in embedded SQL	237
16.1 Using embedded SQL in DB2 UDB applications	238
16.1.1 Components of a DB2 UDB application	238
16.1.2 Creating a package	238
16.1.3 Incorporating prep/bind into make	240
16.1.4 Embedded SQL files as libraries	241
16.2 Multiple connections in embedded SQL programs	241
16.2.1 Connection context	242
16.2.2 Context operations	242
16.2.3 Client-server considerations	249
Chapter 17. Packaging applications for deployment	251
17.1 Creating a project	252
17.1.1 Example source structure	252
17.1.2 Adding prerequisite libraries to the project	254
17.1.3 Prepare the database	254
17.1.4 Customize the application	255
17.2 Creating RPM packages	255
17.2.1 Before you begin	256
17.2.2 Preparing the source archive	256
17.2.3 Preparing package specification	257
17.2.4 Building the package	257
17.2.5 Installing packages	258
17.3 Creating WAR packages	258
17.3.1 Building a WAR package using Ant	259
17.3.2 Deploying WAR packages	262
17.3.3 Deployment on WebSphere Application Server 4.0	263
Part 4. Appendixes	265
Appendix A. DB2 for Linux on zSeries	267
Installing DB2	268
Before you begin	268
Prerequisites	268
Installation procedure	269
Configuring DB2	272
Appendix B. Porting applications to Linux on zSeries	275
Linux for S/390 and zSeries porting - hints and tips	276
Graphics support on the mainframe	280
Memory debuggers	281
Application profilers	281
Porting UNIX applications to Linux: Hints and tips	282
Key questions to consider before starting	282

Will migration involve a huge initial investment.	282
How much will it cost, and how long will it take.	283
Will my application continue to work on the original UNIX platform	283
Porting from Linux to Linux on zSeries	284
Summary.	284
Solaris-to-Linux porting guide.	285
Java applications.	285
Fortran applications.	285
Run-time interfaces.	286
Additional considerations	286
Technical guide for Solaris to Linux application porting	286
Porting overview	286
Use the grep command.	287
Identify potential problems	287
Use a porting tool	287
Appendix C. Tools for Windows workstations	289
PuTTY - a Telnet/SSH client for Windows	290
Cygwin - a UNIX emulator for Windows.	291
Using XFree86 as an X Server under Cygwin	292
Using WebSphere Application Server on Windows	293
Filenames	293
Carriage return/line feed	293
Property files under WebSphere Studio Application Developer	294
Command line completion under Windows.	294
Copying files from Windows to Linux	294
Java Virtual Machine environment	295
Just in Time compiler (JIT)	295
Appendix D. Linux for S/390 VM HONE pilot	297
Introduction.	298
Background	300
HONE/LINK.	300
Green screens.	301
HONWeb/LINKWeb.	302
GWA.	304
Linux for S/390	305
Production rollout	305
System configuration	305
Conclusion	306
Appendix E. Additional material	307
Locating the Web material	307
Using the Web material	307

System requirements for downloading the Web material	308
How to use the Web material	308
Abbreviations and acronyms	309
Related publications	311
IBM Redbooks	311
Other resources	311
Referenced Web sites	312
How to get IBM Redbooks	315
IBM Redbooks collections	315
Index	317

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
AlphaWorks®
DB2 Connect™
DB2®
@server™
Home Director™
Hummingbird®
IBM®
IBMLink™
iSeries™

Lotus®
MQSeries®
MVS™
Notes®
OS/390®
Perform™
RACF®
Redbooks(logo)™ 
S/390®
SP™

VisualAge®
VM/ESA®
WebSphere®
Word Pro®
z/Architecture™
z/OS™
z/VM™
zSeries™

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

Lotus®

Word Pro®

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM Redbook describes application development for Linux on the IBM @server zSeries platform. The target audience is application developers writing primarily in C/C++ and Java. The topics covered will be familiar to programmers familiar with Linux application development on other hardware platforms; we note differences and optimizations specific to the zSeries platform where applicable.

The Linux development environment for zSeries is quite similar to the development environment on other platforms running Linux since the operating system services and development tools share a common code base. Linux on zSeries offers many of the same application languages: C and C++, Java, Perl, and Python, to name just a few.

Programmers writing in a language that provides a runtime environment (such as Java or Perl) are accustomed to the portability that runtime can provide. In fact, Java's platform-neutral implementation is one of its key benefits, and Perl's runtime has been implemented on a wide variety of platforms. Porting C and C++ applications across hardware platforms has typically required more effort and expense. Linux has helped to greatly reduce those porting costs because porting many C and C++ applications simply requires recompilation.

On the zSeries platform, Linux application development can offer some unique advantages. Running Linux images as guests under zVM allows consolidation of development servers on a centrally managed machine, thus simplifying system administration of the development environment. Hardware virtualization provided by zVM allows physical resources to be shared by multiple Linux guests.

This redbook is divided into three parts:

- ▶ In part one, we discuss some standard development tools available for Linux on the zSeries platform. We provide complete details for using the IBM Java Software Development Toolkit, CVS, Emacs, the vi editor, and applications that make up the Jakarta project.
- ▶ In part two, the open source Eclipse Integrated Development Environment is introduced. We describe the basic concepts it incorporates, and provide step-by-step instructions for installing, configuring, and working with Eclipse.
- ▶ In part three, we demonstrate programming techniques using an example J2EE application as an illustration. All the code necessary to implement the sample project in your own environment is included.

An appendix provides details about installing DB2 for Linux on zSeries, presents topics related to porting applications, and describes a pilot project that produced an e-business solution on Linux for S/390.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Gregory Geiselhart is a project leader for Linux on zSeries at the International Technical Support Organization, Poughkeepsie Center.

Andrea Grahn is an I/T Architect for IBM Global Services in Poughkeepsie, New York. She has 5 years of experience in Internet application development. She holds a bachelor of science degree in Computer Science from Polytechnic University in Brooklyn, New York. Her areas of expertise include developing and designing J2EE applications for both IBM internal and commercial customers.

Frans Handoko is a senior IT specialist for IBM Global Services in Uithoorn, the Netherlands. He joined IBM as hardware engineer, then moved to application development, mostly in the VM/CMS platform and recently in the VM/Linux platform as well. He holds a master of science degree in Electrical Engineering from the Technical University of Delft (the Netherlands). His areas of expertise include text retrieval and pricing.

Jörg Hundertmark is an IT Specialist for IBM Global Services in Cologne, Germany. He has 4 years of experience developing e-business applications. He holds a master degree in Computer Science from the University of Paderborn, Germany. His areas of expertise include J2EE Web applications, and he has written extensively on Java topics.

Albert Krzymowski is an Advisory Education Specialist for IBM Global Learning Services in Warsaw, Poland. His area of expertise include DB2 UDB and MQSeries running on all IBM @server branches. He has over 5 years experience in C applications development. He holds a master of science degree in Computer Science from Warsaw University. He has written extensively on C and C++ topics.

Eliuth Pomar is a software engineer for the IBM Server Group in Poughkeepsie, New York. He has 10 years of experience in VM, OS/390, Linux, and DB2 application and system programming. He holds a bachelor of science degree in Information Systems from York College, and a master of science degree in Computer Science from Rensselaer Polytechnic Institute. His areas of expertise

include application development for Linux/390. He has written extensively on Eclipse.

Thanks to the following people for their contributions to this project:

Terry Barthel, Dave Bennin, Ella Buslovich, Alison Chandler, Roy Costa, and Al Schwab
International Technical Support Organization, Poughkeepsie Center

Paul Sutura
IBM Poughkeepsie

Theresa Halloran
IBM Poughkeepsie

Malcolm Zung
IBM Canada

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

ibm.com/redbooks

- Send your comments in an Internet note to:

redbook@us.ibm.com

► Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYJ Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400



Part 1

Programming tools

In this part of the book, we introduce some tools for application development available for Linux on zSeries. Topics discussed include:

- ▶ An assortment of tools traditionally used for C and C++ development
- ▶ IBM Java Software Development Kit for Java development
- ▶ CVS revision control system
- ▶ emacs editor
- ▶ vi editor
- ▶ The Jakarta project, an assortment of Java development tools
- ▶ zSeries architectural consideration, including debugging and optimization

The basic tools you need

In this chapter, we discuss programming tools used in development on Linux for zSeries. We focus mainly on utilities for C and C++ applications; however, you may find some of these topics helpful when you write programs in other programming languages like Java or Perl.

Most of the tools we describe have very good documentation available on the Internet or in Linux distributions. Therefore, we only:

- ▶ Describe the benefits of using a tool
- ▶ Highlight the most useful features
- ▶ Provide simple examples
- ▶ Provide references to the documentation
- ▶ Discuss functions or differences (if any) specific to Linux for zSeries

1.1 Where you can look for information

If you work with most popular Linux distributions, there are four major sources of information available to you:

- Man pages
- Info help systems
- Numerous HOW-TOs
- Books you have received with your Linux CDs

All contain information useful for a programmer. In this section, we briefly describe the first two.

1.1.1 Man pages

Man pages are common to most UNIX systems and have been a main source of information for programmers and users for years. You can invoke the man manual with a **man** command:

```
man [section] keyword
```

For example, type:

```
man man
```

to get to know how to use the manual system. Usually, the content you can see on man pages is presented by a program called more. Press **q** to end it and get back to the command prompt.

Note: Linux provides an improved version of more called less. Yes, the less utility has more features than does more (for example, backward scroll when reading from a pipe). It is advisable to set:

```
alias more=less
```

in your `~/.profile`. The man command uses less if it is available on your Linux distributions.

Often there are more entries for a single keyword. For example, write is a shell command and it is also a system function. In such a case, man displays the first entry it comes across when searching sections in the following order:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within system libraries)
4. Special files (usually found in /dev)

5. File formats and conventions, for example `/etc/passwd`
6. Games
7. Macro packages and conventions (such as `man(7)`, `groff(7)`).
8. System administration commands (usually only for root)
9. Kernel routines (*non-standard*)

For example, if you want information on the system function `write`, type:

```
man 2 write
```

Tip: If you want to search for man pages related to the particular topic, you can use the shell command **`apropos topic`**, where *topic* is a topic to search on. For instance, the command

```
apropos pthread
```

displays all man page titles where *pthread*s are mentioned. The *topic* may be a regular expression.

1.1.2 Info - the help system

Info usually contains more information on a particular topic than `man`. Moreover, it often offers many interesting examples and tips.

You can invoke `info` either as a standalone program or as an Emacs specialized buffer (press `Ctrl - h i` once you have run the text editor).

The information you see can be considered as hypertext (organized in tree-like structure) and viewed in a read-only buffer of a text editor. All Emacs shortcuts and commands can be used to scroll or search through the text (refer to Chapter 4, “The Emacs editor” on page 55 for further details).

You can select a *link* (text after an asterisk, usually in bold face) by placing the cursor at the beginning of the line and pressing `Enter`.

Other useful shortcuts are:

u	Moves up a node
n	Moves to the next
p	Moves to the previous node
Ctrl - x Ctrl- c	Ends a standalone program
Ctrl - x k	Closes an info buffer in Emacs

1.2 Compiling C/C++ source code

In the following section we discuss the GUN collection of compilers. Although we talk mainly about the C compiler, described features apply to other programming languages like: C++, Objective C, and Fortran.

1.2.1 Starting gcc

To produce a standalone program from a collection of source files, start gcc from a command line as follows:

```
gcc -o output_name main.c somefun1.c somefun2.c ...
```

Option:

-o specifies the output file name. If not specified, the default will be a.out.

Source files may be compiled separately:

```
gcc -c module.c
```

This command creates an object file called module.o, which can be incorporated into a:

- Program (one of the modules should contain a function named main with proper declaration):

```
gcc -o program_name module1.o module2.o ...
```

- Static library:

```
ar csr libname.a module1.o module2.o ...
```

- Dynamic library:

```
gcc -shared -o libname.so -Wl,-soname,libname.so module1.o module2.o ...
```

Refer to 14.2.6, “Building shared libraries” on page 202 for more details on how to create libraries in Linux.

1.2.2 Source files

Naming convention for source files are listed below.

*.c	C source code which must be preprocessed
*.i	C source code which should not be preprocessed
*.ii	C++ source code which should not be preprocessed
*.m	Objective-C source code
*.h	C header file (not to be compiled or linked)
*.cxx	C++ source code which must be preprocessed

*.cc	C++ source code which must be preprocessed
*.C	C++ source code which must be preprocessed
*.s	Assembler code
*.S	Assembler code which must be preprocessed

1.2.3 Directory search

The following options specify directories to search for header files and libraries.

- I *directory* Specifies a directory for header files searches.
- I - All directories following this option will be searched for header files specified as #include "*hdr.h*" (#include <*hdr.h*> are not affected.)
- L *directory* Specifies a directory for library file search.

Directories are searched in the order they were specified. Standard directories like /usr/include or /usr/lib are searched at the end.

1.2.4 Compilation stages

A compilation can consist of four stages:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

The steps always appear in this order.

Note that the first three stages apply to an individual source file. Linking combines all the objects into a single executable file.

The following options determine where compilation stops:

- c Compile or assemble the source files, but do not link. The object file name for a source file is made by replacing the suffix '.c', '.i', '.s' with '.o'.
- S Stop after the compilation phase; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified. The assembler file name for a source file is made by replacing the suffix '.c', '.i', etc., with '.s'.
- E Stop after the preprocessing stage and do not run the compiler. This option sends results to standard output or to the specified file.

Tip: You will find this option very useful while tracking syntax errors in macro definitions. You can run gcc with -E to see how the code looks after all macros, includes, or conditional statements like `#ifdef` have been applied.

1.2.5 Macros

Some options relevant to macro pre-processing are:

- D *Name* Define macro *Name* setting its value as 1.
- D *Name=Def* Define macro *Name* setting its value as *Def*.
- U *Name* Undefine macro *Name*. This option is evaluated after all define (-D) options.

1.2.6 Warnings

Some options relevant to compile warnings are:

- Wall Warns about constructions which are questionable or ugly, although satisfying the syntax (compilation continues).

Tip: We strongly advise you to use this option. Often such warnings help find errors related to wrong castings.

- w Inhibit all warning messages (note that the **w** is lowercase).
- Werror Treat all warnings as errors.

1.2.7 Extra information for debuggers

Some options relevant to debugging are:

- g Add debugging information for programs like gdb, dbx. Unlike most other C compilers, gcc allows you to use -g with -O. However, the effects seen during debugging may surprise you.
- ggdb Include GDB extensions if possible.
- p Add extra code to write profile information suitable for the analysis program **prof**. Refer to the notes on profiling in 7.2.4, “Performance profiling” on page 102 and the **gprof** manual in the info system.

Tip: Use the same debug options for modules when compiling a program.

1.2.8 Code optimization

Following are descriptions of general optimization switches. For more details on an optimization process and its consequences, refer to 7.3, “Optimizing for performance” on page 102.

-O or -O1	Turn optimization on.
-O2	Optimize better than -O and improve the output unless it does not lead to the significant conflicts between space and speed factors. Compiler does not perform loop unrolling or function inlining.
-O3	Optimize better than -O2, try to improve execution time.
-Os	Optimize for size. -Os enables all '-O2' optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.
-O0	Do not optimize (uppercase letter 'O' followed by zero).

If you turn on optimization, compilation may take longer and consume more memory (especially when compiling large functions or case statements). Unless explicitly specified, the compiler tries to reduce code size and execution time.

1.2.9 Configuring gcc as a cross-compiler

By default, gcc compiles code for the same type of machine it runs on. However, it may be configured as a cross-compiler. A cross-compiler allows you to create programs that can be run on architectures different from the one used for a compilation. In this section we describe how we compiled and configured the gcc compiler on Linux for zSeries in order to produce binaries for i386.

Cross-compiler configuration consists of the following steps:

1. Downloading and installing prerequisites.
2. Compiling binutils.
3. Installing libraries.
4. Compiling the cross-compiler.

We describe installing the compiler along with utilities in the `/usr/cross-devel/` directory. This simplifies sharing this directory among VM Linux images.

Downloading and installing prerequisites

Make sure to have a full development environment on your machine. At least the following packages should be present:

- ▶ gcc-compiler and standard C/C++ development libraries
- ▶ binutils package

- ▶ autoconf (provides autoheader utility)
- ▶ yacc and flex packages

Compiling binutils

Before you compile the binutils package, you must obtain the source code. We used the `binutils.spm` package provided by SuSE distribution.

1. Install `binutils.spm`. The source tarball will be extracted to the `/usr/src/packages/SOURCE` directory.
2. Unpack it to the temporary directory:


```
bunzip2 < /usr/src/packages/SOURCES/binutils-2.11.90.0.27.tar.bz2 | \
tar -C /tmp -xf -
```
3. Change your working directory to the binutils source and run configure:


```
cd /tmp/binutils-2.11.90.0.27/
./configure --prefix=/usr/cross-devel --target=i386-pc-linux s390-ibm-linux
```
4. Start the compilation:


```
make
```
5. Install the binaries:


```
mkdir /usr/cross-devel
make install
```
6. Now examine the contents of the `/usr/devel-i386/bin` directory:


```
ls /usr/cross-devel/bin/
.                i386-pc-linux-gasp      i386-pc-linux-readelf
..               i386-pc-linux-ld      i386-pc-linux-size
i386-pc-linux-addr2line i386-pc-linux-nm      i386-pc-linux-strings
i386-pc-linux-ar      i386-pc-linux-objcopy i386-pc-linux-strip
i386-pc-linux-as      i386-pc-linux-objdump
i386-pc-linux-c++filt i386-pc-linux-ranlib
```

Installing libraries

In order to produce executable files, you need to install libraries precompiled for a particular architecture. Compiling `glibc` from source is possible, but we suggest using the precompiled version that is commonly used on Linux for i386. You can download `glibc` from:

<http://www.rpmfind.net>

Look for the version closest to that installed on your platform:

```
rpm -q glibc
glibc-2.2.2-25
rpm -q glibc-devel
glibc-devel-2.2.2-25
```

In our example we used version 2.2.2-38 (part of the SuSE Distribution for Linux on i386). We needed both `glibc.rpm` and `glibc-devel.rpm`.

Include files and libraries should be installed in the directories whose names begin with *compiler_prefix/architecture*. In this case we used:

```
/usr/cross-devel/i386-pc-linux/include
```

and

```
/usr/cross-devel/i386-pc-linux/lib
```

These library packages cannot be installed using the `rpm` tool (this would conflict with existing libraries). Instead, use the following procedure:

1. Create a temporary directory:

```
mkdir /tmp/i386
```

2. Unpack libraries with `cpio`:

```
cd /tmp/i386
rpm2cpio ../glibc.rpm | cpio -id
rpm2cpio ../glibc-devel.rpm | cpio -id
```

3. Move the include part to `/usr/cross-devel/i386-pc-linux`. You don't have to create `/usr/cross-devel/i386-pc-linux`. It is created during installation of the `binutils` package as described in step 5 on page 10.

```
cd /tmp/i386/usr
cp -a include /usr/cross-devel/i386-pc-linux
```

4. Move the contents of `lib` and `usr/lib` directories to the `/usr/cross-devel/i386-pc-linux/lib` directory:

```
cd /tmp/i386
cp -a lib /usr/cross-devel/i386-pc-linux
cd usr
cp -a lib /usr/cross-devel/i386-pc-linux
```

5. Merge files from the two directories. Now you have to fix some symbolic links in `/usr/cross-devel/i386-pc-linux/lib`.

```
cd /usr/cross-devel/i386-pc-linux/lib
```

6. All you need to do is remove the prefix `../../lib/` from symbolic links. Be careful typing. This is a single command:

```
ls -l | grep ../../lib | cut -c57- | cut -d' ' -f1,3 | \
sed 's^../../lib/^' | \
while read a b ; do rm -f $a ; ln -s $b $a ; done
```

7. Correct `libc.so` accordingly (this is a plain text file):

```
/* GNU ld script
Use the shared library, but some functions are only in
```

```

        the static library, so try that secondarily. */
GROUP (
    /usr/cross-devel/i386-pc-linux/lib/libc.so.6
    /usr/cross-devel/i386-pc-linux/lib/libc_nonshared.a
)

```

Compiling the cross-compiler

The `gcc.spm` package that comes with SuSE Linux 7.2 for zSeries contains the main source `gcc-2.95.3.tar.gz` and patches specific for S/390. Look for:

- `gcc-2.95.3-s390.tar.gz`
- `gcc-2.95.3-s390-1.tar.gz`
- `gcc-2.95.3-s390-2.tar.gz`

Use the following steps to prepare the source code.

1. Unpack the main source package:

```
tar -C /tmp -zxf /usr/src/packages/SOURCES/gcc-2.95.3.tar.gz
```

2. Unpack patches in the temporary directory:

```

mkdir /tmp/gcc-patches
cd /tmp/gcc-patches
tar -zxf /usr/src/packages/SOURCES/gcc-2.95.3-s390.tar.gz
tar -zxf /usr/src/packages/SOURCES/gcc-2.95.3-s390-1.tar.gz
tar -zxf /usr/src/packages/SOURCES/gcc-2.95.3-s390-2.tar.gz

```

3. Apply the patches:

```

cd /tmp/gcc-2.95.3
patch -p1 < /tmp/gcc-patches/gcc-2.95.3-s390.diff
patch -p1 < /tmp/gcc-patches/gcc-2.95.3-s390-1.diff
patch -p1 < /tmp/gcc-patches/gcc-2.95.3-s390-2.diff

```

4. Configure the gcc as a cross compiler:

```

./configure --prefix=/usr/cross-devel \
--with-headers=/usr/cross-devel/i386-pc-linux/include/ \
--target=i386-pc-linux s390-ibm-linux

```

Note: We have found that the `--with-headers` option is essential in order to avoid having to specify the `-I` option when working with the cross-compiler later on.

5. Compile:

```
make
```


Note: In some configurations, the `PATH_MAX` constant is not defined properly. Add the following lines after the `include` section if you encounter this problem.

```
#ifndef _POSIX_PATH_MAX
#define _POSIX_PATH_MAX 255

#endif
```

6. Install package:

```
make install
```

Using cross compiler

Once the compiler has been installed, you should update your `PATH` environment variable:

```
export PATH=/usr/cross-devel/bin:$PATH
```

Executable files for the i386 platform are created when `gcc` is invoked with a proper prefix. In this example, use the `i386-pc-linux-gcc` command. We suggest setting the `CC` variable to this command to instruct `make` what compiler is to be used.

For example, to make executable `test` from source file `test.c`, issue:

```
$ export CC=i386-pc-linux-gcc
$ make test
i386-pc-linux-gcc    t.c  -o t
$ file test
test: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked
(uses shared libs), not stripped
```

To build applications for i386 (or other platforms) on Linux zSeries, you should install all needed libraries as described in “Installing libraries” on page 10. Cross compilation of libraries is also possible. We recommend you use the same version that is available on the target machines.

1.3 Linking object code

The following options can change behavior of the linker when you compile your code with `gcc`.

Note: Remember that if any of the `-c`, `-S`, or `-E` options are specified, the linker is not invoked and the following options are meaningless.

Some relevant options are:

- | | |
|-------------------|---|
| -s | Remove all symbol table and relocation information from the executable. Same as the <code>strip</code> command. |
| -static | Link with the static libraries. |
| -shared | Produce a shared object that can then be linked with other objects to form an executable. Refer to 14.2.6, “Building shared libraries” on page 202. |
| -Wl,option | Pass <code>option</code> as a parameter to the linker. |

1.4 Automating the build process

Most projects consist of a number of files that are compiled separately and then, if required, linked together into standalone programs or libraries. If you change the source file, you should recompile all other files which depend on that file. If it is a header file, you should recompile all files in which it is included. On the other hand, if you change a C source file (*.c), only this file needs to be recompiled and linked.

The `make` tool checks the timestamps on your source files and invokes compilation only if necessary. `Make` operates on a configuration file (commonly referred to as a *Makefile*).

The default configuration file names are `makefile` and `Makefile`.

1.4.1 GNU make

The `make` command included in Linux distributions is a GNU version of `make`. It is compliant with POSIX.2 standards. However, there are a lot of extensions, especially regarding implicit rules. Sophisticated makefiles may not work with standard `make` utilities supplied with systems other than Linux. In such a case you will have to adjust a makefile or install `gmake` (GNU version of `make`).

1.4.2 Writing your Makefile

In the following example, input for the `make` tool will produce a program called `sample`. It consists of two modules compiled separately (line 14 and 17) and linked together (line 11). The interface for functions exported in the `somefun` module is provided in a header file called `somefun.h`. Both modules will be recompiled whenever this file is changed.

Example 1-1 Makefile rules

```
01:  # Sample Makefile
02:
03:  INCS    =      -I/usr/lib/qt-2.3.0/include -I ../itsodb
04:  LIBS    =      -lpthread -ldl
05:
06:  OBJS     =main.o somefun.o
07:
08:  CFLAGS   = -g $(INCS)
09:
10:  sample: $(OBJS)
11:      gcc -o sample $(OBJS) $(LIBS)
12:
13:  main.o:  main.c somefun.h
14:      gcc $(CFLAGS) -c main.c
15:
16:  somefun.o: somefun.c somefun.h
17:      gcc $(CFLAGS) -c somefun.c
18:
19:  clean:
20:      rm -f *.o *~
```

Now we take a closer look at Makefile syntax.

Comments

line 01 Text on the right of the # sign is considered to be a comment.

Variables

lines 03-08 We define some useful variables here. Values associated with variables can be accessed by the following expression:
 \$(variable_name).

line 08 The definition can have a reference to another variable.

Note: Variable values can be set in the shell. If you set shell variable CC, make may use its value as a command in rules like this one:

```
$(CC) $(CFLAGS) mymodule.c
```

Rules

lines 10-20 A single rule for a make tool has the following format:

```
<target>: <prerequisites>
<TAB><command 1>
<TAB><command 2>
```

<target> Name of the file to created

- <prerequisites> List of the file names that must exist before the commands are executed. List items are separated by spaces. If the list is too long to fit into the line, you can end it with the backslash sign and continue in the next one.
- <command .> Shell command to be executed. Note that the single <TAB> is essential to satisfy Makefile syntax.

Rules are used to:

- Define the commands that are to be invoked to produce a target
- Decide whether the target is out of date

Rules may be generic; and you specify patterns. The following example shows how the make tool produces object files (*.o) from C source code (*.c) when there is no explicit definition.

This pattern is predefined in the GNU implementation of make:

```
% .o : % .c
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Some of the automatic variables are:

- \$@ File name of the target of the rule
- \$% Target member name
- \$^ Names of all the prerequisites, with spaces between them
- \$< Name of the first prerequisite

1.4.3 Building with make

When make is invoked, you can specify the target you want to build. By default, the first one is created (and of course, all other files that are necessary). To make target myecho, issue:

```
make myecho
```

Conventions

When preparing a Makefile for your project, you should consider the following targets:

- all** Provide an all rule and place it at the beginning. This rule should define the default target.
- clean** This rule tells make how to remove all unnecessary files (like *.o objects and temporary entries).

- install** This rule tells make how to install your project. Look at the **Make -> Makefile Conventions -> Directory Variables** in the Linux info help system for some hints on how to define this rule.
- depend** If your project consists of many files, you will probably find creating dependencies very confusing. If your project is compiled for another target platform, header files may be located in other places and the compilation may not succeed. Use the `makedepend` utility to prepare the list of dependencies and append that list to the end of Makefile.

1.4.4 Makedepend

The `makedepend` program reads each of the source files specified as an argument and looks for all include statements. Information on every file that was included is stored in Makefile as a dependency:

```
sourcefile.o: dfile1 dfile2
```

In this example `sourcefile.o` is the name from the command line with its suffix replaced with `.o` and `dfile1` is a dependency discovered in the `#include` directive while parsing `sourcefile` or one of the files it includes.

By default, the output is placed at the end of Makefile, after the line which looks like this:

```
# DO NOT DELETE THIS LINE -- make depend depends on it
```

All subsequent calls will replace the content below this message.

`makedepend` respects other C-preprocessor statements such as `#define` and `#ifdef`. You should invoke the program with the options

- D...** For define statements
- I...** For include directories

in addition to any compiler options you supply to `gcc`.

1.4.5 File dependencies from gcc

The `gcc` compiler may be used to generate dependencies as well. Use the following options:

- M** Tells the preprocessor to output a rule suitable for make
- MM** Like `'-M'` but the output uses only the user header (files included with `#include "FILE"`). System header files using `#include <FILE>` are omitted.

1.5 Libraries

Libraries provide functions and data constructs that can be used in your programs. Since an end user needs different library packaging than a programmer, several kinds of archives are usually available:

- ▶ Binary distribution - shared objects for use with compiled programs. Distributed as *libname.rpm*.
- ▶ Include files for library objects, static version of a library and other necessary files you need to build your program. Distributed as *libname-devel.rpm*.
- ▶ Library source files. Distributed as *libname.spm* or *libname.src.rpm*
- ▶ Precompiled version of a library with settings for profiling or debugging. Distributed as *libname-profile.rpm*

Refer to 14.2.5, “Using shared libraries” on page 200 for details about how to use libraries.

Standard C and C++ libraries

SuSE Linux Enterprise Server 7 contains packages with *libg++-libc6.2-2* libraries.

- ▶ Include files are in *gpp.rpm*
- ▶ Shared objects are in *gppshared.rpm*

Some of the precompiled programs (including IBM DB2 UDB v.7.1) require an older version. You might see an error like this:

```
libstdc++libc6.1-2.so.3:cannot load shared object file: No such file or
directory
```

You can find the missing files on SuSE 7.2 for S390 distribution in a package named in exactly the same way: *gppshare.rpm*. Although it provides the 2.95.2-165 version of the library, we cannot install it due to rpm version rules.

The *rpm2cpio* tool can be used instead, using the following steps.

1. Download the *gppshare.rpm* into a temporary directory.
2. Unpack the library.

```
mkdir /tmp/gppshare
cd /tmp/gppshare
rpm2cpio /tmp/gppshare.rpm | cpio -di
```

3. Copy the files you need.

```
cp -a /tmp/usr/lib/libstdc++* /usr/lib
```

What libraries are available?

An up-to-date list of the libraries available on Linux for zSeries is available at:

http://www.ibm.com/servers/eserver/zseries/os/linux/ldt/slate_enablers.html

For more information on libraries go to the following site:

<http://www.rpmfind.net>

Even if you do not find the precompiled version for zSeries, the link to source code is often provided. Download the source package and build it with the `rpm` command.

```
rpm --rebuild packagename.src.rpm
```

Be sure to install all prerequisite software in advance. Note that this command does not install the package, but it builds the `rpm` file and stores it in the `/usr/src/packages/RPM/s390` directory. This name may vary depending on your architecture and Linux distribution.

1.6 Tracking changes

In this section we show how two Linux utilities, `diff` and `patch`, can be used to distribute new versions of software.

1.6.1 Using `diff` to find differences

The `diff` command shows differences between two files. Running `diff` on text files generates a line by line report on differences (for binary files, `diff` will only report that the files differ). This output is often called a *diff*. For identical files, no report is generated.

Note: There is also a `diff3` command that compares three input files and lets you inspect two different sets of changes to the same file. Emacs offers an `emerge-files` function that let you to incorporate two tiers of changes into a single file.

As an example, we use two versions of the simple program `echo`. Example 1-2 shows the first version of the `echo.c` program.

Example 1-2 echo.c

```
01:  #include<stdio.h>
02:
03:  main(int argc, char *argv[]){
04:
```

```
05:    int i;
06:
07:    i=1;
08:
09:    while(i<argc)
10:        printf(argv[i++]);
11:
12:    printf("\n");
13: }
```

The improved version, where you can suppress printing an end of line character, is presented in Example 1-3,

Tip: We used the `getopt` function to intercept the `-n` switch. Refer to `man 3 getopt` for more information on `getopt` (a shell command called `getopt` also exists to deal with parameters in shell scripts).

Example 1-3 echo2.c

```
01:  #include<stdio.h>
02:  #include<unistd.h>
03:
04:  main(int argc, char *argv[]){
05:
06:      int i;
07:      int no_nl;
08:
09:      no_nl = getopt(argc,argv,"n") == 'n';
10:      i=optind;
11:
12:      while(i<argc)
13:          printf(argv[i++]);
14:
15:      if(!no_nl)
16:          printf("\n");
17:  }
```

Let's have a look at how the `diff` command works. To compare the two files, type in:

```
diff echo1.c echo2.c > echo.diff
```

We intercept the text sent to standard output and store it in the separate file named `echo.diff`. Example 1-4 on page 21 shows the result. The line numbers are provided for convenience - they are *not* part of the `diff` output.

Example 1-4 *echo.diff*

```
01: 1a2
02: > #include<unistd.h>
03: 5a7
04: > int no_nl;
05: 7c9,10
06: < i=1;
07: ---
08: > no_nl = getopt(argc,argv,"n") == 'n';
09: > i=optind;
10: 11,12c14,16
11: < printf("\n");
12: ---
13: > if(!no_nl)
14: >     printf("\n");
```

The diff discovered that:

- Two lines are added (01-02,03-04),
- There are also two changes (05-09,11-14).

Notice that **diff** does not get lost, and matches `printf` instructions properly. The the line with `printf` instruction is noted as changed due to the extra space added as an indentation following the `if` statement.

1.6.2 Applying changes

Provided the output from the **diff** is placed in the `echo.diff` file, changes may be applied to the `echo1.c` file using the **patch** command:

```
patch echo1.diff < echo.diff
```

Now there should be no difference between `echo1.c` and `echo2.c`.

Even if you modify your `echo1.c` program slightly (for example, by adding the following to line 6 in the main function):

```
printf("Welcome to echo example\n");
```

patch is likely to update the source code properly. It always tries to anchor the changes somewhere within context.

1.6.3 Running diff against source tree

In practice, the procedure described previously cannot be used to apply patches for a new software version for at least two reasons:

- ▶ Only one file was updated, not the whole source tree.
- ▶ The **patch** command easily gets lost if significant changes are made.

For example, if we had deleted the `printf("\n");` statement in line 12, the tail of the new version proposed by **patch** looks like:

```
while(i<argc)
    if(!no_nl)
        printf(argv[i++]);
```

This is definitely not the version we would like to see. In the next two sections, these problems are addressed.

Recursive comparison

You can run **diff** against directories. When **diff** arguments are directories, comparisons are made against all files contained in both directories (examining files in alphabetical order).

Some options related to this kind of operation are:

- r** Compares corresponding pair of files in the directory, stepping down into subdirectories if necessary. By default, **diff** reports subdirectories common to both directories without comparing files (equivalent to the `--recursive` option).
- s** Reports pairs of identical files. By default, **diff** prints nothing on files that contain no differences.
- N** If a file is found in only one directory, treat it as present but empty in the other directory.
- P** This option is similar to **-N** except it only inserts the contents of files that appear in the second directory but not the first (**patch** describes only the files that were added). This may reduce the size of your patch, but use it with caution.

Context patch

If you want to distribute new versions of your files in the form of a **diff** patch, you should use one of context output formats. The **patch** tool can apply the diffs in this case by searching in the files for the lines of context around the differing lines. If those lines are not far away from where the **diff** says they are, the **patch** can adjust the line numbers accordingly and carry on its work.

Options related to the context output are:

- c** Use the context output format that shows several lines around the the fragments that differ.
- u** Use the unified output format. This is a more compact version of the context patch. You can also switch this format on with **-U**.
- U *lines*** Similar to **-u**, but the argument *lines* specifies the number of lines of context to show. By default, it is set to three.

Let's get back to our example and move the old version of the echo program to the directory echo-v1, naming it echo.c. The new version is moved to directory echo-v2 and is also named echo.c.

Now we allow the context output and invoke the diff command to run recursively:

```
diff -Nua echo-v1 echo-v2 > echo.patch
```

The obtained output is shown on Example 1-5.

Example 1-5 echo.patch

```
01: diff -Nua echo-v1/echo.c echo-v2/echo.c
02: --- echo-v1/echo.cWed May  8 16:23:05 2002
03: +++ echo-v2/echo.cWed May  8 16:23:10 2002
04: @@ -1,13 +1,17 @@
05:  #include<stdio.h>
06:  +#include<unistd.h>
07:
08:  main(int argc, char *argv[]){
09:
10:      int i;
11:      + int no_nl;
12:
13:      - i=1;
14:      + no_nl = getopt(argc,argv,"n") == 'n';
15:      + i=optind;
16:
17:      while(i<argc)
18:          printf(argv[i++]);
19:
20:      + if(!no_nl)
21:          printf("\n");
22:      }
```

The **diff** uses the following markers:

- +** Stands for an inserted line

- Stands for a deleted line
- ! Stands for a line that is part of a group of one or more lines that were changed (not shown in this example)

We suggest the following invocation of the `diff` command when the patch for a new version of source code is being produced:

```
diff -Nur new_dir old_dir > project-version.patch
```

You should always check whether the patch applied against the old version produces a new one without any rejects. If not, add a few more lines of context, to help the patch tool:

```
diff -Nur -U5 new_dir old_dir > project-version.patch
```

1.6.4 Distributing patches

The patch tool takes comparison output produced by `diff` and applies the differences to a copy of the original file, producing a patched version. With `patch`, you can distribute just the changes to a set of files instead of distributing the entire file set.

The patch tool automatically determines the `diff` format, skips any leading or trailing headers, and uses the headers to determine which file to patch. It detects and warns about common problems like forward patches and, in such a case, saves the original version of the file as well as any patches that could not be applied.

If you have a patch for a new version of the source directory, look at its contents and find the `diff` command that has been used. In the next two lines after the `diff` invocation you will find the names of the files to be patched (lines number 02 and 03 in Example 1-5).

In most cases, names of your directories differ from those specified in the patch. You can deal with it with the following options:

- | | |
|---------------------------|---|
| <code>-pnumber</code> | Sets how many slashes (along with the directory names between them) are to be stripped from the front of file names. Option <code>-p</code> with no number given is equivalent to <code>-p0</code> . By default, <i>patch</i> strips off all leading directories, leaving just the base file names. |
| <code>-d directory</code> | Sets <i>directory</i> as the current directory for interpreting both file names in the patch file. |

Let's get back to our `echo.c` example. We upgrade the original source (`~/src/echo/echo.c`) using the patch (`/tmp/echo.patch`) as follows:

```
patch -d ~/src/echo -p1 < /tmp/echo.patch
```

We use the `-p1` option to strip the directory names `echo-v1` and `echo-v2` from the file names.

Tip: Sometimes people run `diff` using the new file as the first file name. If you get such a *reversed* patch, apply it with the `-R` or `--reverse` option.

1.6.5 Before you distribute your patch

When you distribute a patch:

- ▶ Always write a short info indicating how it should be applied. Be sure to mention the prerequisites.
- ▶ Remove all unnecessary files from your directories before creating a patch.

If you provide users with source code for your application (that is, you work on an open source project), new versions should provide both a full distribution and a patch to be applied against the previous version whenever possible. One advantage of sharing source code is the fact that users can modify the application simply by changing the source. Users that have modified your source (to customize the code or to fix bugs) will appreciate your diff. These changes can be merged into the new version if you supply a patch.

The IBM Java Software Development Kit

This chapter briefly describes Java as an application platform and discusses:

- ▶ The Java 2 platform
- ▶ The IBM Java Software Developer Kit for Linux
- ▶ The IBM Java SDK
- ▶ The Jikes Java compiler

2.1 Java 2 Platform, Software Development Kit

The Java 2 Platform is distributed with the Software Development Kit (SDK). The Java 2 Platform SDK (also referred to as the J2SDK) provides development tools such as:

- Java compiler
- Java debugger
- Java documentation builder

2.1.1 References

More information on Java, including the documentation for the J2SE and J2EE specifications, can be found at the Sun Microsystems Web site for Java located at:

<http://java.sun.com>

2.2 IBM Java Developer Kit for Linux running on zSeries

Sun does not provide a Java 2 implementation for Linux running on zSeries. However, as part of its commitment to the Java platform, IBM has ported the J2SDK implementation to zSeries Linux. For a complete list of platforms supported by the IBM Java SDK, see:

<http://www.ibm.com/developerworks/java/jdk/>

2.2.1 Obtaining the IBM Java Developer Kit

This redbook is based on version 1.3.1 of the IBM J2SDK. This version is freely available for download at:

<http://www.ibm.com/developerworks/java/jdk/linux130/>

The RPM version (we used file `IBMJava2-SDK-1.3.1-1.0.s390.rpm`) should be selected for easy installation.

2.2.2 Installing the IBM Java Developer Kit

As root, install the package using the command:

```
rpm -ivh IBMJava2-SDK-1.3.1-1.0.s390.rpm
```


By default, rpm will install the J2SDK in the directory /opt/IBMJava2-s390-131. This will be the Java home directory. An outline of the package contents is presented in Table 2-1.

Table 2-1 Contents of the IBM J2SDK package

Directory	Purpose
/opt/IBMJava2-s390-131/	Java home directory - also contains source code for IBM J2SDK
/opt/IBMJava2-s390-131/bin	Executable directory - contains compiler, debugger, archiver, applet viewer
/opt/IBMJava2-s390-131/demo	Example code directory - contains various ready-to-run Java demonstrations
/opt/IBMJava2-s390-131/docs	Documentation directory - contains license and readme files
/opt/IBMJava2-s390-131/include	Include directory - contains include files for Java native interface (JNI)
/opt/IBMJava2-s390-131/jre	Runtime directory - contains shared libraries and executables for JRE
/opt/IBMJava2-s390-131/lib	Archive directory - contains Java archive (JAR) for J2SDK

2.3 Jikes

Developed by the IBM T.J. Watson Research Center, Jikes is a Java command line compiler like the standard Java compiler javac. The Jikes compiler was released in binary form on the IBM AlphaWorks site in 1997. Jikes for Linux followed in July 1998. Now the source code is available under IBM's Public License, which has been approved by the Open Source Initiative (OSI) as a fully certified open source license. Officially it is no longer supported by IBM; instead, it survives today solely on free contributions from the open source community.

Jikes offers some advantages over the standard Java compiler. For example, Jikes provides:

- ▶ Better performance (Jikes is written in C++)
- ▶ Better error messages
- ▶ Dependency analysis, which can be used for incremental builds and Makefile generation

The Jikes home page is located at:

<http://www.ibm.com/developerworks/oss/jikes/>

For Linux on zSeries, we had to take the source rpm file (available in the download section) and build the binary ourselves.

2.3.1 Installing Jikes

Here are the steps to install Jikes in Linux:

1. Download the `jikes-1.15.-1.src.rpm` file from the Jikes site.
2. In the download directory, issue the commands:

```
rpm --rebuild jikes-1.15-1.src.rpm
rpm -ivh /usr/src/packages/RPMS/s390/jikes-1.15-1.s390.rpm
```

Note that the second command requires root rights.

3. Add Jikes to the CLASSPATH:

```
export CLASSPATH=$CLASSPATH:/opt/IBMJava2-s390-131/jre/lib/rt.jar
```

You must have a version of the JDK or JRE to run Jikes because the compiler needs to access the standard class files. Note that since version 1.1 of the JDK it is no longer necessary to provide a definition of CLASSPATH to run `javac` or `java`. However, `jikes` does not know what version of the JDK you are using and therefore you must indicate where it should find the standard library files.

2.3.2 Using Jikes

Invoke Jikes in the form:

```
jikes [ options ] filename
```

Invoke Jikes without arguments to get a summary of the options. Some options in Jikes that are not found with `javac` are:

++	Compile in incremental mode.
+E	List errors in a form commonly used by Emacs to scan for errors. By default, errors are listed in a more readable form.
+M	Generate makefiles with dependencies.

Jikes can compile more than one file at a time, for example:

```
jikes Test1.java Test2.java Test3.java
```

Jikes also accepts arguments starting with an at sign (`@`). Such arguments are taken to be the name of a file, with each line then processed as though it were itself an argument, except that lines starting with `@` are not processed recursively. For example, the above command could also be written as:

jikes @file.list

where the file `file.list` contains the lines:

```
Test1.java  
Test2.java  
Test3.java
```

See the Jikes home page for more details.

Source code control using CVS

In this chapter we discuss CVS, an open source version control system for project file management. Some of the basic features will be demonstrated; many more are available. For a full explanation of CVS and all its features, refer to the CVS manual *Version Management with CVS* by Per Cederqvist, et al., available at:

<http://www.cvshome.org/docs/manual/>

3.1 Introduction to CVS

CVS operates on a client-server model, allowing developers in widely dispersed geographic locations to collaborate on a common software project. CVS relies on the **diff** command to determine version differences. By recording version differences and applying the **patch** command to previous revisions, it is possible to recover any file revision stored in CVS. As it relies on the **diff** and **patch** commands to recreate a file history, CVS works best on text files.

Tip: Do not check in CVS files generated during the build process (particularly binary files such as object code, byte-compiled Java class files, or binary executable files). Rather, check in text files created by developers (such as source files, configuration files, and Makefiles), and allow the defined build process to generate the binaries. The binaries can then be packaged along with other CVS-maintained files (using the **tar** command, for instance) for application deployment.

3.1.1 Definitions

Before going into details on CVS operation, some terms should be defined.

Repository	The copy of all files and directories under CVS version control.
Module	A directory contained within a repository. Repositories may contain multiple modules. Modules are used to group files in the repository.
Working copy	The local copy of the module on which developers make modifications.
Check out	The action to obtain a working copy.
Commit	The action to reflect changes made to a working copy back to the repository.
Update	The action to incorporate changes made and committed to a repository module back into a working copy. Updates are required when more than one developer works concurrently on a module.
Revision number	A unique number for each file, used to identify committed revisions.

3.1.2 Revision numbering

Each file stored in the repository is assigned its own revision number. Revision numbers take the form of a series of period-separated digits. By default, CVS will assign 1.1 to the first revision of a file. As revisions are made to a file, increases are made to the last digit in the revision. For instance, revision numbering proceeds from 1.1 to 1.2 to 1.3. It is possible to assign a specific revision number to a file, although normally it is simpler to allow CVS to maintain revision numbering.

3.1.3 File locking

Unlike some source code control systems, CVS does not by default rely on reserved checkouts to maintain version integrity. (Reserved checkout implies that only a single developer is allowed to edit any file; this is typically enforced by file locking in the version code system.)

Instead, CVS allows multiple editors to operate on any given file in their respective working copy. The first editor to commit changes will succeed in getting their changes reflected into the repository. When other editors attempt to commit their changes, they will receive an error message indicating their working copy is out of date with respect to the repository. At that point, it is possible to incorporate the previously committed revision into the working copy without losing changes (see 3.13, “Resolving conflicts” on page 51).

It is possible to get CVS to perform reserved checkouts, but this feature is rarely used and will not be discussed here. For details, see the CVS manual.

3.2 CVS command syntax

The syntax for CVS commands is as follows:

```
cvs [ global_opts ] command [ cmd_opts ] [ args ]
```

where:

cvs
global_opts
command
cmd_opt
args

is the **cvs** program name
are global options passed to the **cvs** command
is the sub-command to execute
are options passed to the sub-command
are arguments passed to the sub-command

3.2.1 Global options

In general, global options specify behavior common to all sub-commands, such as the level of verbosity generated on stdout. Most of the global options can be set as exported shell variables, or as entries in the `~/.cvsrc` file (see 3.6, “Environment variables and the `~/.cvsrc` file” on page 40). Some of the more commonly used global options are listed in Table 3-1.

Table 3-1 Common CVS global options

Option	Description
<code>-T tempdir</code>	Use <i>tempdir</i> as the location for temp files; overrides the <code>\$TMPDIR</code> environment variable.
<code>-d cvsroot</code>	Specifies root directory of CVS repository; overrides the <code>\$CVSR00T</code> environment variable.
<code>-e editor</code>	Use <i>editor</i> to enter log information; overrides the <code>\$CVSEEDITOR</code> and <code>\$EDITOR</code> environment variables.
<code>-f</code>	Do not read <code>~/.cvsrc</code> file.
<code>-n</code>	Execute command but do not add, remove, or update files.
<code>-Q</code>	Very quiet mode; report only serious errors.
<code>-q</code>	Quiet mode; informational messages are not reported.

3.2.2 CVS commands

Command names indicate the action CVS is to take. For instance, the command **add** indicates CVS is to add files from the working copy to the repository. Some commands have aliases that can be used in place of the command name. Some of the more commonly used commands are listed in Table 3-2.

Table 3-2 Common CVS commands

Command	Description
init	Initialize a repository.
checkout	Create or update a working copy (alias: co and get).
add	Add files or directories to repository (alias: ad and new).
commit	Commit changes to repository (alias: ci).
diff	Show differences between revisions.
import	Import files to repository.

Command	Description
log	Print log information found in repository.
status	Display the status of files in working relative to repository (alias: st and stat).
update	Bring working copy up to date with respect to repository (alias: up and upd).

3.2.3 Command options

Commands may take additional options in addition to global options. In general, command options are specific to each command. Additionally, not all command options are supported by every command, and some command options conflict with global options. The `cvs` man should be consulted for details. Some of the more commonly used command options are listed in Table 3-3.

Command options are options passed to the specific command. Unfortunately, some command option names conflict with global option names, the global option having a completely distinct meaning from the command option meaning. Consult the man pages for reference.

Table 3-3 Common CVS command options

Option	Description
-D <i>date</i>	Specify a revision no later than <i>date</i> (see 3.2.6, “Date formats” on page 38).
-l	Operate only in current working directory - do not act recursively.
-m <i>message</i>	Use <i>message</i> as log entry (see 3.2.5, “Log messages” on page 38).
-P	Prune empty directories.
-p	Pipe files to stdout rather than write to working copy.
-R	Recursively descend directories (this is the default behavior).
-r <i>tag</i>	Use <i>tag</i> to identify revision on which to operate.

3.2.4 Command arguments

Arguments are also specific to commands. In general, arguments refer to file names or module names. By default, CVS commands operate recursively in the current working directory. This means that when executing the command with no file arguments in the root directory of a working copy, `cvs` will operate on all files in that directory as well as files in all sub-directories.

3.2.5 Log messages

Commands which change revisions to files in the repository require messages to be supplied as command options (these messages become part of the revision history and may be examined). To specify a log message, use the `-m` message option. As usual, surrounding a message with double quotes (") enables shell variable expansion, surrounding a message with single quotes (') disables shell variable expansion. If no `-m` option is supplied, cvs will invoke an editor to prompt the user for message text.

3.2.6 Date formats

CVS accepts dates expressed in many formats. The standard ISO8601 date formats as well as the standard e-mail formats (RFC822 and RFC1123) are recognized. Dates are interpreted to be expressed in the local timezone unless a timezone is explicitly included. CVS tolerates partial date formats and even dates expressed as offsets for the current time. Be aware that if a date does not specify a timezone, the timezone specified on the local machine will be used. Some examples of valid dates are:

- 13 May 2001 21:00 GMT
- 13 May 2001
- 13 May
- 3 days ago
- 2 hours ago

Do not forget to quote the date when supplied as an option to `-D`.

Tip: While accepted as valid, date formats of the form *month/day/year* can be confusing (some people use this form as *day/month/year*). Use another date format to avoid any confusion.

3.3 Administering CVS

The first step in CVS administration is to ensure the code has been installed on all machines requiring access to the repository. Most distributions provide CVS packaged as an rpm file, usually located in the development section. For SuSE 7.0 SLES, CVS is packaged as the `cvs.rpm` file found on CD 1.

3.3.1 Creating a repository

Once CVS is installed, it is necessary to create the repository. After deciding where in the file system the repository is to reside, issue the following command as root:

```
cvs -d /var/cvs init
```

where /var/cvs is the directory path chosen to hold the repository.

Administrative files

On initialization, a directory named CVSR00T and containing administrative files is created in the repository. The files inside this directory are used to control the behavior of CVS. These files should not be edited directly; they are under CVS revision control.

Setting file permissions

Once a repository is created, file ownership and permissions should be set. Create a group for all developers requiring access to the repository (group **cvs** for example), add developers to that group, and change group ownership and permissions on repository, as illustrated in Example 3-1.

Example 3-1 Creating a CVS repository

```
$ cvs -d /var/cvs init
$ cd /var/cvs
$ chgrp -R cvs .
$ chmod -R o-rwx .
$ chmod u+rwX . CVSR00T
$ chmod g+rwXs . CVSR00T
```

Enabling remote access

Upon creation, the repository is immediately accessible (subject to UNIX file permissions) to developers on the local machine. CVS supports a variety of access methods for remote repositories. Remote repositories are identified by the form of the root directory name for the repository specified on the cvs command (see the -d global option in Table 3-1).

3.4 Root directory

The CVS repository root directory is specified in the form:

```
:method:[ [ user ][:password]@host:[ port ]/repository-root
```

where:

<i>method</i>	Specifies the access method
<i>user</i>	Connect using userid
<i>password</i>	Password for userid
<i>host</i>	Host on which the repository resides

<i>port</i>	Port to connect over
<i>repository-root</i>	Full path to the CVS repository

In the case where only a repository root directory is specified, CVS recognizes the repository as residing locally. In general, the password parameter should not be specified (to prevent cleartext passwords from exposure). Leaving it out will cause CVS to prompt for it on the command line. To simplify the process of defining the repository root directory, CVS accepts the value of environment variable CVSROOT as a default. Typically, developers will define this variable in their login profile and not specify the -d global CVS option.

The access method is described in the next section. For more details, consult the CVS reference manual.

3.5 ssh access

Using remote shell execution, the CVS client can perform actions on the repository. The default remote shell for CVS is **rsh**. Due to security considerations, **rsh** is typically disabled on most Linux machines. However, CVS supports **ssh** as an **rsh** replacement. To enable ssh access, an ssh server must be running on the remote repository machine and an ssh client must be installed in the local developer machine. Each developer must also have a login userid defined on the remote repository machine (this being the userid and password defined in the root directory specification). For rsh and ssh access, the method is defined as **ext3**. To distinguish between rsh and ssh, the environment variable CVS_RSH should set to the value ssh (**export CVS_RSH=ssh**).

3.6 Environment variables and the ~/.cvsrc file

Environment variables affect the behavior of CVS—typically in providing default values to global options. Some of the more widely used environment variables are detailed in Table 3-4.

Table 3-4 Common environment variables used by CVS

Variable Name	Usage
CVSROOT	Specifies the repository root directory (see “Root directory” on page 39); overridden by the -d global option.
CVS_RSH	Specifies the :ext: access method; defaults to rsh .

Variable Name	Usage
\$EDITOR \$CVSEDITOR \$VISUAL	Specifies the editor to use when providing log messages for commit . Use the -m global option to provide log messages from the command line.

Tip: Developers typically export \$CVSROOT and \$CVS_RSH (when using the ssh access method) in their login shell profile. This eliminates the need to continually provide a long **-d** option string on cvs commands.

To alleviate the need to continually provide default options to commands, CVS will read the `~/.cvsrc` file to add defaults on a per-command basis. For instance, if the **-P** (prune) option is always to be supplied to the checkout command, an entry such as

```
checkout -P
```

may be added to the `~/.cvsrc` file. Default options are specific to each command. So, to add a default **-l** option to the commit command, the `~/.cvsrc` file would appear as:

```
checkout -P
commit -l
```

If a command has an alias, the actual command name should be specified in the file.

3.7 Creating a project

The first step in creating a project is to start with a local directory structure (or at least as much as is known at the beginning of the project) that matches the structure to be saved in the repository. Add files to the project in the appropriate place in the directory structure.

3.7.1 Importing the files

Once the local directory is populated, the CVS import command can be used to bring the project into the repository. Example 3-2 illustrates creating a simple project consisting of the following files:

- Makefile
- src/file1.c
- include/hdr1.h

Example 3-2 Creating a new project using CVS import

```
$ echo "$CVSR00T $CVS_RSH" 1
:ext:myuser@cvshome:/var/cvs ssh
$
$ mkdir -p proj/src proj/include 2
$ cd proj
$ echo "initial revision" > Makefile
$ echo "initial revision" > src/file1.c
$ echo "initial revision" > include/hdr1.h
$ echo "delete this line" >> include/hdr1.h
$
$ cvs import -m'initial import' proj myuser start 3
myuser@cvshome's password:
N proj/Makefile
cvs server: Importing /var/cvs/proj/src
N proj/src/file1.c
cvs server: Importing /var/cvs/proj/include
N proj/include/hdr1.h

No conflicts created by this import

$ cd .. 4
$ mv proj proj-save
$
```

1. The `$CVSR00T` and `$CVS_RSH` variables indicate the repository is remote with the ssh access method.
2. The project initially consists of files `Makefile`, `src/file1.c`, and `include/hdr1.h`.
3. Project is imported to directory `/var/cvs/proj` on remote repository. The *myuser* argument specifies a vendortag, *start* specifies a releasetag. Typically a userid and character string (such as *start*) are used. Later, both files and directories may be added to the repository (see 3.10, "Adding files and directories" on page 45).
4. Directory is renamed to permit a working copy to be checked out in this directory.

3.8 Obtaining a working copy

Before operating on a repository, it is necessary to check out a local copy on which to edit - the *working copy*. Continuing with the example begun in previously, the command sequence illustrated in Example 3-3 will check out a working copy.

Example 3-3 Checking out a working copy

```
$ cvs co proj 1
myuser@cvshome's password:
cvs server: Updating proj
U proj/Makefile
cvs server: Updating proj/include
U proj/include/hdr1.h
cvs server: Updating proj/src
U proj/src/file1.c
$
$ cd proj
$ ls 2
CVS Makefile include src
$
$ ls CVS
Entries Entries.Log Repository Root 3
$
```

1. Check out module `proj`. A working copy will exist in the `proj` sub-directory.
2. List files in the working copy.
3. CVS directory contains special files.

3.8.1 Special files

In the working copy, CVS creates a special directory named `CVS` that is used to assist CVS in managing the working copy. These files should not be edited. Some of these files are shown in Table 3-5; refer to the manual for complete documentation.

Table 3-5 CVS administrative files in the working copy

File	Purpose
Entries	Lists each file and directory in the working copy.
Entries.Log	A copy of the Entries file.
Repository	Records the directory in the repository from which the working copy is derived.
Root	Records CVS root directory (<code>\$CVSR00T</code>).

Important: The name `CVS` is reserved as both a file and directory name by CVS. Do not name any file or directory `CVS`.

3.9 Making changes in the working copy

Changes are made to files in the working copy. CVS is able to distinguish modifications to the working copy relative to the repository using the **cvs diff** command (see 1.6.1, “Using diff to find differences” on page 19). Note the recursive behavior of CVS commands exhibited in Example 3-4.

Example 3-4 CVS tracks changes made to working copy relative to repository

```
$ sed s/initial/next/ Makefile > temp; mv temp Makefile 1
$ sed '/delete this line/d' include/hdr1.h > temp; mv temp include/hdr1.h
$ echo 'added this line' >> src/file1.c
$
$ cvs diff 2
myuser@cvshome's password:
cvs server: Diffing .
Index: Makefile
=====
RCS file: /var/cvs/proj/Makefile,v 3
retrieving revision 1.1.1.1
diff -r1.1.1.1 Makefile
1c1
< initial revision
---
> next revision
cvs server: Diffing include
Index: include/hdr1.h
=====
RCS file: /var/cvs/proj/include/hdr1.h,v 4
retrieving revision 1.1.1.1
diff -r1.1.1.1 hdr1.h
2d1
< delete this line
cvs server: Diffing src
Index: src/file1.c
=====
RCS file: /var/cvs/proj/src/file1.c,v 5
retrieving revision 1.1.1.1
diff -r1.1.1.1 file1.c
1a2
> added this line
$
```

1. A line is modified in Makefile, deleted from include/hdr1.h, and added to src/file1.c.
2. Use **cvs diff** to report differences in the working copy with respect to the repository.

3. Modification to line 1 in the Makefile (1c1) is noted.
4. Deletion of line 2 in include/hdr1.h (2d1) is noted.
5. Addition of line 2 in src/file1.c (1a2) is noted.

3.10 Adding files and directories

Next, an additional directory (etc) will be added to the repository. Then files will be added to that directory and to the src directory. Note that before files can be added to a directory, the directory must be added to the repository. The sequence is illustrated in Example 3-5.

Example 3-5 Adding files and directories to working copy

```
$ mkdir etc 1
$ echo 'initial revision' > etc/proj.conf
$ echo 'initial revision' > src/file2.c
$
$ cvs add -m'add etc/ dir' etc 2
myuser@cvshome's password:
? etc/proj.conf
Directory /var/cvs/proj/etc added to the repository
$
$ cvs add -m'add etc/proj.conf src/file2.c' etc/proj.conf src/file2.c 3
myuser@cvshome's password:
cvs server: scheduling file `etc/proj.conf' for addition
cvs server: scheduling file `src/file2.c' for addition
cvs server: use 'cvs commit' to add these files permanently
$
```

1. A new etc directory is created in the working copy, new files are added to etc and src directories.
2. The etc directory is added to the repository.
3. The etc/proj.conf and src/file2.c files are added to the repository.

Note that the **cvs add** command takes a required file specification argument list.

Important: Adding a file *will not* cause the file to be added to the repository—it simply schedules the file for addition. To complete the action, it is necessary to issue the **cvs commit** command. Unlike files, directories are added immediately to the repository.

3.11 Committing changes to the repository

To complete file additions as well as file revisions to the repository, it is necessary to execute the **cvs commit** command, as illustrated in Example 3-6.

Example 3-6 Committing changes to the repository

```
$ cvs -Q status 1
myuser@cvshome's password:
=====
File: Makefile          Status: Locally Modified 2

    Working revision:    1.1.1.1
    Repository revision: 1.1.1.1 /var/cvs/proj/Makefile,v

=====
File: proj.conf         Status: Locally Added 3

    Working revision:    New file!
    Repository revision: No revision control file

=====
File: hdr1.h           Status: Locally Modified 4

    Working revision:    1.1.1.1
    Repository revision: 1.1.1.1 /var/cvs/proj/include/hdr1.h,v

=====
File: file1.c          Status: Locally Modified 5

    Working revision:    1.1.1.1
    Repository revision: 1.1.1.1 /var/cvs/proj/src/file1.c,v

=====
File: file2.c          Status: Locally Added 6

    Working revision:    New file!
    Repository revision: No revision control file

$
$ cvs ci -m'commit #1' 7
cvs commit: Examining .
cvs commit: Examining etc
cvs commit: Examining include
cvs commit: Examining src
myuser@cvshome's password:
Checking in Makefile;
/var/cvs/proj/Makefile,v <-- Makefile
new revision: 1.2; previous revision: 1.1
```

```
done
RCS file: /var/cvs/proj/etc/proj.conf,v
done
Checking in etc/proj.conf;
/var/cvs/proj/etc/proj.conf,v <-- proj.conf
initial revision: 1.1
done
Checking in include/hdr1.h;
/var/cvs/proj/include/hdr1.h,v <-- hdr1.h
new revision: 1.2; previous revision: 1.1
done
Checking in src/file1.c;
/var/cvs/proj/src/file1.c,v <-- file1.c
new revision: 1.2; previous revision: 1.1
done
RCS file: /var/cvs/proj/src/file2.c,v
done
Checking in src/file2.c;
/var/cvs/proj/src/file2.c,v <-- file2.c
initial revision: 1.1
done
$
etc/proj.conf src/file2.c
```

1. The **cvs status** command is issued to verify only intended changes are committed. The file status field is explained in Table 3-6.
2. The file `Makefile` is noted as Locally Modified.
3. The file `etc/proj.conf` is noted as Locally Added.
4. The file `include/hdr1.h` is noted as Locally Modified.
5. The file `src/file1.c` is noted as Locally Modified.
6. The file `src/file2.c` is noted as Locally Added.
7. Revisions are committed using the **ci** alias for the **cvs commit** command. Note the command's recursive behavior.

Important: To avoid committing unintended revisions, always issue the **cvs status** or **cvs diff** command before **cvs commit**. Check command output to ensure only files you wish to commit are changed before proceeding.

Note: When a file is imported using **cv**s **import**, it is assigned an initial revision number of 1.1.1.1. On the first commit, that revision number is changed to 1.1. The 1.1.1.1 revision can be viewed as equivalent to revision number 1.1.

3.12 Updating the working copy

When collaborating on a project, revisions committed by one developer act to make the working copy of other developers out of date with respect to the repository. The **cv**s **update** command is used to bring a working copy up to date.

In general, two types of updates are possible:

- ▶ An update from the repository that does not conflict with the revision present in the working copy
- ▶ An update that does conflict with the working copy

Non-conflicting updates occur when no modifications have been made to the file in the working copy: CVS can simply replace or apply a patch to the file. Conflicts occur when the file has been modified in the working copy: changes present in the repository must then be merged into the working copy while preserving the local modifications.

CVS is capable of recognizing and dealing with both types of updates. The **cv**s **status** is used to identify files requiring updates. Table 3-6 lists file status states as reported by **status**. The **cv**s **update** command is used to update to the working copy.

Table 3-6 CVS working copy status states

Status state	Meaning
Up-to-date	File in working copy is current relative to repository.
Locally Modified	File has been modified in working copy but not yet committed.
Locally Added	File is new in working copy but not yet added to repository.
Locally Removed	File has been deleted in working copy but not yet committed.
Needs Checkout	File has a newer revision in repository than exists in working copy.

Status state	Meaning
Needs Patch	Like Needs Checkout, but rather than replacing the working copy file, CVS can apply a patch to bring file up to date.
Needs Merge	File has a newer revision in repository and modifications have been made to working copy.
File had conflicts on merge	CVS has merged revisions into working copy. Conflict needs to be manually resolved.
Unknown	File exists in working copy but is unknown to repository; the cvs add command needs to be executed on the file.

Example 3-7 illustrates a situation where the working copy of one user (*myuser*) requires an update due to revisions committed by another user (*otheruser*). Revisions to the file `Makefile` are non-conflicting in this working copy, while revisions to the file `include/hdr1.h` conflict in this working copy.

Example 3-7 Updating the CVS working copy

```

$ echo "added by myuser" >> include/hdr1.h      1
$
$ cvs -Q status                                  2
myuser@cvshome's password:
=====
File: Makefile      Status: Needs Patch          3

    Working revision: 1.2
    Repository revision: 1.3    /var/cvs/proj/Makefile,v

=====
File: proj.conf      Status: Up-to-date

    Working revision: 1.1
    Repository revision: 1.1    /var/cvs/proj/etc/proj.conf,v

=====
File: hdr1.h         Status: Needs Merge          4

    Working revision: 1.2
    Repository revision: 1.3    /var/cvs/proj/include/hdr1.h,v

=====
File: file1.c        Status: Up-to-date

    Working revision: 1.2
    Repository revision: 1.2    /var/cvs/proj/src/file1.c,v

```

```

=====
File: file2.c          Status: Up-to-date

    Working revision:   1.1
    Repository revision: 1.1    /var/cvs/proj/src/file2.c,v

$
$ cvs update
myuser@cvshome's password:
cvs server: Updating .
P Makefile
cvs server: Updating etc
cvs server: Updating include
RCS file: /var/cvs/proj/include/hdr1.h,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into hdr1.h
rcsmerge: warning: conflicts during merge
cvs server: conflicts found in include/hdr1.h
C include/hdr1.h
cvs server: Updating src
$
$ cat include/hdr1.h
initial revision
<<<<<< hdr1.h
added by myuser
=====
added by otheruser
>>>>>> 1.3
$

```

-
1. Revisions are made to the file `include/hdr1.h` in the working copy.
 2. The **cvs status** command is issued prior to commit.
 3. The file `Makefile` is in status `Needs Patch`, indicating another user has committed revisions to the file since the update of this working copy.
 4. The file `include/hdr1.h` is in status `Needs Merge`. This indicates another user has committed revisions to the file and the file has been modified in the working copy of myuser. This has resulted in a conflict which needs to be resolved before this file can be committed by myuser.
 5. To resolve the conflict and to get the latest revision of `Makefile`, the **cvs update** command is executed.
 6. CVS applies a patch to `Makefile` to bring it up to date.
 7. Revisions to the file `include/hdr1.h` are merged into the working copy. CVS notes the conflict.

8. Examination of `include/hdr1.h` reveals the line added by `otheruser` that is causing the conflict.
9. Line added by CVS to indicate start of revisions made in working copy.
10. Line added by CVS to indicate end of revisions made in working copy. Lines that follow are revisions merged from the repository.
11. Line added by CVS to indicate end of revisions merged from repository (1.3).

Tip: To reduce the possibility of conflicts, it is best to have clearly assigned ownership of project files among developers.

Important: Because `cvs update` brings revisions from the repository into the working copy, it is important to ensure only operational revisions are committed to the repository. Incomplete or untested changes should never be committed because they can easily break the working copy for all developers.

3.13 Resolving conflicts

To resolve the conflict identified in Example 3-7, it is necessary to edit the file. The CVS indicators must be removed, and revisions merged in the correct order. The result is shown in Example 3-8.

Example 3-8 Resolving CVS conflicts

```
$ cat include/hdr1.h
initial revision
added by otheruser
added by myuser
$
```

3.14 Viewing log messages

Log messages provided on commit may be viewed using the `cvs log` command. Example 3-9 shows the log file for `include/hdr1.h`.

Example 3-9 CVS message log

```
$ cvs log include/hdr1.h
myuser@cvshome's password:

RCS file: /var/cvs/proj/include/hdr1.h,v
Working file: include/hdr1.h
```

```

head: 1.4
branch:
locks: strict
access list:
symbolic names:
    start: 1.1.1.1
    myuser: 1.1.1
keyword substitution: kv
total revisions: 5;    selected revisions: 5
description:
-----
revision 1.4
date: 2002/05/10 15:31:53; author: myuser; state: Exp; lines: +1 -0
commit by myuser
-----
revision 1.3
date: 2002/05/10 15:23:13; author: otheruser; state: Exp; lines: +1 -0
otheruser revision
-----
revision 1.2
date: 2002/05/10 15:00:37; author: myuser; state: Exp; lines: +0 -1
commit #1
-----
revision 1.1
date: 2002/05/10 14:47:28; author: myuser; state: Exp;
branches: 1.1.1;
Initial revision
-----
revision 1.1.1.1
date: 2002/05/10 14:47:28; author: myuser; state: Exp; lines: +0 -0
initial import
=====
$

```

3.15 Recovering versions

CVS allows for recovery of any file revision using **cvs update**. In Example 3-10, revision 1.1 of file `include/hdr1.h` is recovered.

Example 3-10 Recovery of a prior CVS revision

\$ cat include/hdr1.h	1
initial revision	
added by otheruser	
added by myuser	
\$	
\$ cvs update -p -r 1.1 include/hdr1.h > include/hdr1.h	2


```
myuser@cvshome's password:
=====
Checking out include/hdr1.h
RCS:  /var/cvs/proj/include/hdr1.h,v
VERS: 1.1
*****
$ cat include/hdr1.h
initial revision
delete this line
$
```

3

1. File contents before revision recovery.
2. Using **cvs update** to recover revision 1.1 (-p option indicates output to be piped to stdout, -r 1.1 indicates revision 1.1 is to be recovered).
3. File contents after revision recovery.

The Emacs editor

This chapter describes how to use Emacs as an integrated tool for writing and building programs. We discuss the features of Emacs that may be helpful in developing both C/C++ and Java applications.

4.1 Editing files using Emacs

In this section we introduce the basic features of the Emacs editor.

4.1.1 Starting Emacs

Depending on the display you work with, Emacs will execute in one of two modes. On a text-based terminal, the Emacs display occupies the whole screen; on the X Windows System, Emacs opens its own windows. The benefit of using Emacs in the second mode is that you can take advantage of the scroll bars, mouse, copy-and-paste clipboard, and additional frames¹.

Type `emacs` to start the editor.

Tip: If Emacs does not start in the mode you expect, check the `DISPLAY` environment variable. If set, Emacs will open a window on the display pointed to by this variable. Otherwise the text editor will run in the text mode and occupy the whole terminal from which it was called. Refer to “Using XFree86 as an X Server under Cygwin” on page 292 for further details on the X Windows environment.

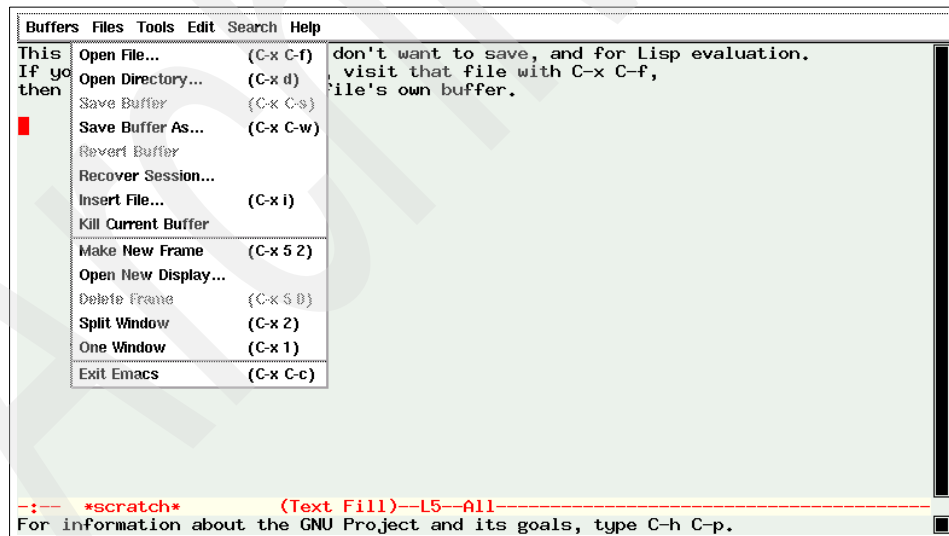


Figure 4-1 Emacs screen in an X Windows environment

¹ We use the term *frame* to mean an entire text screen or an entire X Windows used by Emacs, and *window* to describe the area where a buffer is displayed.

4.1.2 Basic commands

Once the Emacs is started you can open existing files or create new ones.

The basic shortcuts are described as follows:

Ctrl-x Ctrl-f	Open a file or create a new one. You can also edit remote files accessible through an ftp server. Use the following convention: /user@hostname:path You will be asked for your password on the remote machine. (also Files -> Open File)
Ctrl-x Ctrl-s	Save the current file. (also Files -> Save Buffer)
Ctrl-x Ctrl-w	Save the current file under a new name. (also Files -> Save Buffer As)
Ctrl-x S	Save all files.
Ctrl-x Ctrl-c	Exit Emacs. (also Files -> Exit Emacs)

Note: Most of the Linux distributions define basic navigation keys (like PgUp, PgDown, cursor) for PC displays properly. If you came across problems while trying to get them working, remember that you can always use the corresponding Ctrl-<key> sequences described in the *GNU Emacs Reference Card* available at:

<http://www.stanford.edu/group/dcg/leland-docs/emacs.html>

4.1.3 Invoking Lisp functions

Unlike other text editors, Emacs does not assign meanings to keys directly. Since Emacs has a built-in Lisp interpreter, most editing commands are written in Lisp and identified by the function name. There are three ways you can invoke such a function:

1. Call it explicitly.

Press Meta-x. M-x should appear in the minibuffer (*Meta-key* is commonly defined as LeftAlt on a PC keyboard; if it does not work, press Esc-x).

Enter the function name in the minibuffer (for example, replace-string) and press Enter.

While you type something in the minibuffer you can use the Tab key to let Emacs complete the name. If there is more than one function that matches the typed prefix, you will be presented with all possible options. Moreover,

recently used functions are remembered (as well as their arguments, such as the string to be found) and easily accessible with cursor up and down keys.

You can use Emacs help to find out what a particular function does. Press Ctrl-h d and enter the function name.

2. Type in a Ctrl-<key> sequence.

Press Ctrl-h b to see the list of all available key bindings.

Tip: To cancel the function that works in minibuffer (like open-file) you can press Ctrl-g at any time.

3. Click on a desired option in the menu at the top of the frame.

4.1.4 Editing multiple files

You can edit more than one file in an Emacs session.

Concept of buffers

Each time you create a new file or open an existing one, Emacs assigns a new buffer to hold the contents. Many files may be opened at the same time and each of them will be stored in its own buffer.

To select the buffer you want to work with, press Ctrl-x b and type the buffer name. Remember that Emacs maintains the list of the most frequently used names and you can access this list with cursor keys.

You can also pick the buffer from the list with these steps:

1. Press Ctrl-x Ctrl-b and the list of all available buffers is displayed.
2. Pick the buffer by placing the cursor on the beginning of the corresponding line and press Enter.

Although most of the buffers are created by visiting the file, there are some specialized ones, as follows:

scratch	Used for notes you do not want to save or for Lisp evaluation.
Buffer List	Contains all buffers and files names.
info	Contains emacs help system. When you activate this function (M-x info) it works in its own buffer.
compilation	Contains output from the compilation.

If you work in Emacs for a while, you may have a large number of buffers opened. To close some of them, use any of the following:

Ctrl-x k Enter	Close (kill) the current buffer.
Ctrl-x k <buffer name> Enter	Close the buffer you have specified.
Meta-x kill-some-buffers	To be asked, for each buffer, whether it should be closed.

Note: Emacs has many interesting buffers like, for example, those which appear when you play tetris (M - x tetris) or talk to psychiatrist (M - x doctor).

Working with multiple windows

When you start Emacs without parameters, it pops up a frame that shows only one buffer at a time (see Figure 4-1 on page 56). To see more than one buffer at the same time (or two different fragments of the same buffer), do the following:

- ▶ If you work with X Windows display and you want to open the new frame:
Click **Files -> Make New Frame**
- ▶ If you want to divide an existing window, press:
 - Ctrl-x 2 To split into two windows, one above the other
 - Ctrl-x 3 To split into two windows, positioned side by side

Adjust the size of the split part by dragging the status line (X Windows system) or press Ctrl-^ to resize the current window.

Ctrl-x o Moves the cursor to the next visible window within the frame (including minibuffer).

To close windows within the frame, press:

Ctrl-x 0 Close the active window.

Ctrl-x 1 Close all windows in the selected frame except the selected one.

4.1.5 Moving text

Emacs makes it easy to move text around a buffer.

Selecting text

Some Emacs commands operate only on a part of the text, so before you invoke them, you should select this fragment. We use the term *region* to refer to a selected area. How you specify the region of interest depends on your environment:

- ▶ When using X Windows, either:
 - Click the left mouse button and hold it down while dragging across the region.
 - Click the left mouse button to place the cursor at one end of the region, move the mouse pointer to the other end and click the right mouse button.
- ▶ When using a keyboard:
 - a. Place the cursor at one end of the region.
 - b. Press Ctrl-<space> to set “the mark”.
 - c. Move the cursor to the other end.
 - d. There is no special mark for the second end of the region. One end is always referred to by the position of the cursor. Therefore, call the function you want immediately after you have established the region, or press Ctrl-g to cancel. Press Ctrl-x Ctrl-x to interchange the mark and the cursor.

When you select text with a mouse, the highlighted background disappears after you click to move the cursor, but Emacs (unlike most of the editors) still remembers the fragment. You can paste it with a middle mouse button click (even to a different application), or use the yank command (Ctrl-y) described in the next section.

Other useful commands related to regions are:

Meta-x append-to-buffer	Appends region to contents of specified buffer.
Meta-x copy-to-buffer	Copies region into specified buffer, deleting old contents.
Meta-x insert-buffer	Inserts contents of specified buffer into current buffer, at point.
Meta-x append-to-file	Appends region to contents of specified file, at the end.

Copy and paste

Probably the easiest way to move fragments of a program to a different place is to cut the lines with a `kill` command and insert them with `yank`.

Ctrl-k	Kills all the text on the right of the cursor line, leaving it blank. To kill an entire non-blank line, go to the beginning of the line and press Ctrl-k twice.
Ctrl-y	Inserts the last killed text in place before the cursor (y stands for yank).

4.1.6 Search and replace

Emacs is supplied with several functions that allow you to search for a string and replace it with a new one. Most of them work with regular expressions, too.

One commonly used function is an incremental search:

Ctrl-s incremental search forward (*isearch-forward*).

Ctrl-r incremental search backward (*isearch-backward*).

Immediately after you start it, Emacs reads from your keyboard and places the cursor at the first occurrence of the characters that you have typed. You can move to the next (or previous) occurrence by pressing Ctrl-s (Ctrl-r) once again. Press the Delete key to remove the most recently typed character. Enter exits the function, leaving the cursor at the last visited location.

The search and replace function is often used in conjunction with regular expressions, as follows:

Meta-x replace-regexp <Enter> REGEXPR <Enter> NEWSTRING <Enter>

If you want to refer to the matched string of REGEXPR in NEWSTRING use \&. In the following example we instruct Emacs to take numbers into parenthesis.

Meta-x replace-regexp <Enter> [0-9]+ <Enter> (\&) <Enter>

See emacs help (Meta-x info or Ctrl-h i) for further details on regular expressions.

4.1.7 Modes

Depending on the sort of file you edit, Emacs provides alternative customized *modes*. Each mode can have different key bindings and rules for how to format and highlight the text. The least specialized mode is *Fundamental mode*. It has no specific redefinitions or variable settings, so each Emacs command behaves in the most general manner, and options are in default states. If you want to edit text of a specific type that Emacs knows about, such as C code or English text, you should switch to the appropriate mode, such as *c-mode* or *text-mode*.

Whenever you open a file or write it under a new name, Emacs determines the file type according to its suffix. If the extension is known, the proper mode for this file will be activated. The current mode is shown between parentheses on the status line (see Figure 4-1 on page 56).

If you edit files with non-standard extension (like .sqc for DB2 Embedded SQL), you have to switch to the desired mode manually (c-mode in this case):

Meta-x **c-mode** for C-style highlighting and key bindings.

Add the following lines to the configuration file `.emacs` located in your home directory for permanent settings and restart Emacs:

```
(setq auto-mode-alist
  (cons '("\\.sqc" . c-mode) auto-mode-alist))
```

Whenever you open or create a file with a suffix `.sqc`, Emacs will turn on c-mode automatically.

4.2 Building applications using emacs

Emacs is also a development environment.

4.2.1 Editing program files

When you enter a specialized mode, Emacs helps you write nicely formatted code and (what is more important) avoid syntax errors in your programs. It saves a lot of time during compilation.

Whenever you are in C, C++, Java, or Lisp mode:

- ▶ Text syntax is highlighted accordingly. You can toggle this feature on and off with `Meta-x font-lock-mode`.
- ▶ Code is indented by the text editor. The default settings are reasonable, and if you do not like Emacs spacing you can customize it, but *do not* turn this feature off. It is very helpful. Emacs indents your code properly and pays attention to parentheses, semicolons and declarations.

To have Emacs indent a line, place the cursor there and press `Tab`. To save CPU cycles, Emacs does not start indentation automatically.

Tip: If you find Emacs indentation odd (especially lines moved too far to the left), check your code for syntax errors. Look for missing semicolons and unbalanced parentheses.

If you load a file that was not indented according to current emacs setting, or make significant changes in your code (for example, remove an enclosing brace brackets block), you have to invoke this function explicitly:

- | | |
|----------------------------|--|
| <code>Ctrl-c Ctrl-q</code> | Indents the current top-level function definition or aggregate type declaration (<code>c-indent-defun</code>). |
| <code>Meta-Ctrl-\</code> | Indents all lines in the region (<code>indent-region</code>). |

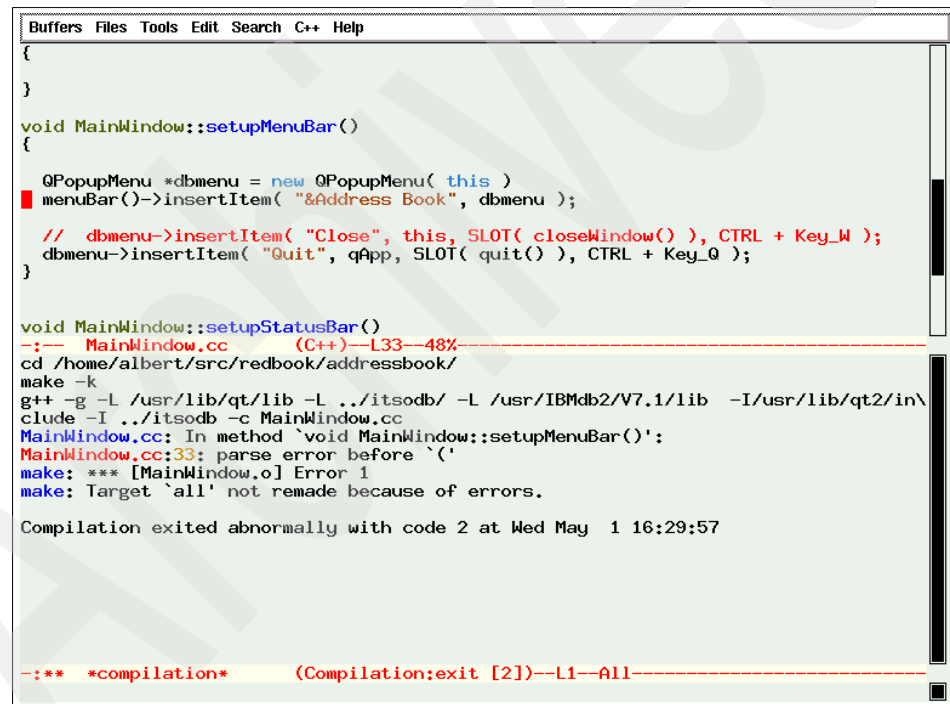
4.2.2 Compiling your application

Many programmers use Emacs not only as an editor but as a fully integrated environment. In this section we show how to build programs and correct compilation errors.

In most popular distributions, a Lisp function **compile** does not have a predefined shortcut. We suggest adding a permanent key binding to your configuration file (`~/.emacs`). The following line sets `Ctrl-c c` as a sequence for compilations.

```
(global-set-key "\C-cc" 'compile)
```

Once the compilation is started you will be asked for a command you would like to be invoked. The default is **make -k** with no arguments. You can change the parameters for a **make** tool or call your compiler explicitly. Emacs stores the command you have entered and this will be a default during subsequent calls.

The screenshot shows an Emacs window with a menu bar (Buffers, Files, Tools, Edit, Search, C++, Help) and a text area containing C++ code and compilation output. The code defines a QMainWindow with a menu bar and a status bar. The status bar setup calls a method 'void MainWindow::setupMenuBar()' which is defined above. The compilation output shows the command 'g++ -g -L /usr/lib/qt/lib -L ../itsodb/ -L /usr/IBMdb2/V7.1/lib -I/usr/lib/qt2/in\clude -I ../itsodb -c MainWindow.cc' being executed, followed by a parse error in the 'void MainWindow::setupMenuBar()' method. The output ends with 'make: Target `all' not remade because of errors.' and 'Compilation exited abnormally with code 2 at Wed May 1 16:29:57'. A special buffer '*compilation*' is shown at the bottom, indicating the compilation has exited with code 2.

```
Buffers Files Tools Edit Search C++ Help
{
}

void MainWindow::setupMenuBar()
{
    QPopupMenu *dbmenu = new QPopupMenu( this )
    menuBar()->insertItem( "&Address Book", dbmenu );

    // dbmenu->insertItem( "Close", this, SLOT( closeWindow() ), CTRL + Key_W );
    dbmenu->insertItem( "Quit", qApp, SLOT( quit() ), CTRL + Key_Q );
}

void MainWindow::setupStatusBar()
-:-- MainWindow.cc (C++)--L33--48%-----
cd /home/albert/src/redbook/addressbook/
make -k
g++ -g -L /usr/lib/qt/lib -L ../itsodb/ -L /usr/IBMdb2/V7.1/lib -I/usr/lib/qt2/in\
clude -I ../itsodb -c MainWindow.cc
MainWindow.cc: In method `void MainWindow::setupMenuBar()':
MainWindow.cc:33: parse error before `('
make: *** [MainWindow.o] Error 1
make: Target `all' not remade because of errors.

Compilation exited abnormally with code 2 at Wed May 1 16:29:57

-:*** *compilation* (Compilation:exit [2])--L1--All-----
```

Figure 4-2 Compiling with Emacs

The compilation is started as a background process and its output is sent to a special buffer `*compilation*`. This buffer tells you whether compilation is finished (watch the word in parenthesis). Note that by default the contents of this window is not scrolled automatically as new lines appear.

Even before the compilation is finished, Emacs starts to examine the log file and lets you go through the errors. You can visit the file and jump to the line with an error by either:

- ▶ Clicking on it with the middle mouse button.
- ▶ Scrolling through the file to the error with the Lisp function **next-error**.

Since it is very inconvenient to call this function with Meta-x **next-error**, we defined a new shortcut (Ctrl-c Ctrl-n) in our configuration file:

```
(global-set-key "\C-cn" 'next-error)
```

Ctrl-u Ctrl-x starts scanning from the beginning of the compilation log.

To kill compilation at any time, enter Meta-x **kill-compilation**. This function should be issued only when you want to stop running processes. If the compilation is finished, it is sufficient to close this buffer.

The vi editor

This chapter introduces the venerable vi editor, which comes with UNIX and with the standard Linux distribution. Actually, in Linux we use vim (vi Improved).

We describe the three operating modes of vi, how to switch from one mode to another, how to customize vi, and we list the most useful commands in vi.

5.1 First encounter with vi

Consider the initial screen that you get when you create a new file by entering:

```
vi newfile.txt
```

(See Example 5-1.)

If you are totally new to vi, you will probably get stuck on that screen, since vi seems to ignore all your input, and you cannot even move the cursor. The reason you cannot move the cursor is that the file is still empty. The tilde (~) represents an empty line, and vi will not allow you to move the cursor to an empty line. The best you can do here is to type `i`, vi will then be receptive to you.

Example 5-1 vi initial screen

```
"newfile.txt" [New File]
```

5.2 Modes of operation

It is important to know that vi operates in three modes:

Command mode	You enter commands in this mode.
--------------	----------------------------------

Insert mode	You insert characters at the cursor position.
-------------	---

Replace mode

You overwrite the character at the cursor position.

To switch from one mode to another:

- ▶ From Command mode to Insert mode, press one of these single characters: i, I, a, A, o, O.

Note: Typing a single character (of the listed set) immediately brings vi to Insert mode. Do not make the mistake of pressing Enter after that. If you do, it will be treated as input in the Insert mode.

- ▶ From Command mode to Replace mode, press R (again, without Enter).
- ▶ From Replace or Insert mode to Command mode, press Esc.

Figure 5-1 illustrates the three modes in which vi operates and the keys needed to go from one mode to the other.

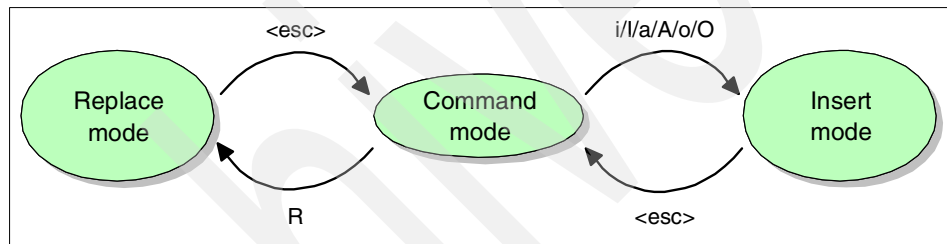


Figure 5-1 vi operating modes

Check the bottom left corner of the vi screen for an indication of the operating mode. There are three possibilities:

- ▶ -- REPLACE --
- ▶ -- INSERT --
- ▶ blanks for Command mode

Normally vi starts in Command mode, and in the initial screen the bottom left corner displays the filename until the first input is entered.

Some utilities like PuTTY make vi easier to use. With PuTTY, for example, it is possible to switch to Insert mode from the other two modes. The Insert key will also bring you from Insert mode to Replace mode.

5.3 Customizing vi

vi has a number of switches you can use to customize the session. These two are particularly useful:

- :set number** Use this to display line numbers. A lot of vi commands depend on line numbers.
- :set ic** Ignore the case when locating a string.

You may want to put the two statements in a file called `.vimrc` in your home directory; these switches will then be activated with each vi session.

In cases when you do not want the line numbers to prefix each record, specify **:set nonumber** to get rid of them. Likewise use **:set noic** to respect the case when locating a string.

To view the file without changing it, it is safest to use the read-only option like this:

```
vi -R filename
```

5.4 Commands categorized by functionality

This section does not pretend to be an exhaustive reference, but it contains a collection of the most frequently used vi command.

Important: Do not terminate a command with Enter unless it starts with a colon (:). The locate commands (Appendix 5.4.3, “Locating a string pattern” on page 69) also need Enter.

5.4.1 Moving the cursor

The commands listed in Table 5-1 move the cursor as shown.

Table 5-1 *Cursor movement*

Command	Moves the cursor to
0	The beginning of the current line
\$	The end of the current line
Enter	The beginning of the next line
w	The beginning of the next word
G	The first non-blank position of the last line

Note: When working under PuTTY you may prefer the Home key to 0 and the End key to \$.

5.4.2 Insertion point

Remember that you can switch from command mode to insert mode. First, place the cursor and then type one of the following single-character commands; this will establish the insertion point as specified:

Command	Insertion point
i	Before the current character
I	Before the beginning of the current line
a	After the current character
A	After the end of the current line
o	At the beginning of a new line that will be inserted before the current line
O	At the beginning of a new line that will be inserted after the current line

After the command is entered the cursor will appear at the insertion point.

5.4.3 Locating a string pattern

Command	What will be located
/xxx <Enter>	Pattern xxx
/ <Enter>	Pattern xxx of a previous /xxx command
?xxx <Enter>	Pattern xxx (search direction is reversed towards the top of the file)
? <Enter>	Pattern xxx of a previous /xxx or ?xxx command

It is possible to use a so-called “regular” expression in the string pattern. For example [0-9] represents any digit and [a,e,i,o,u] represents any vowel.

5.4.4 Replacing

rc	Replace the current character by c
:s/ppp/qqq/	In the current line, replace the first occurrence of pattern ppp by qq
:s/ppp/qqq/g	In the current line, replace all occurrences of pattern ppp by qq
cw	(See the detailed explanation that follows.)

While editing programs, you frequently have to change a word (or part of it). This is where the **cw** (change word) command will come in handy.

Suppose you want to change “equals” in line (1) to “startsWith” as in the line (3). While in Command mode, place the cursor under the e of “equals,” then enter **cw**. The second line will now appear, with the cursor under the second left parenthesis. You will notice that the word “equals” has disappeared and the cursor is positioned to accept insertions for a new word in the place of the word that you want replaced (vi is now in Insert mode). At this point type “startsWith” and press Esc, you will then get line (3).

1. `if (command.equals(prefix))`
 [^]
2. `if (command.(prefix))`
 [^]
3. `if (command.startsWith(prefix))`

Important: In vi, words are delimited by space, but also by any character that is not a digit and not a letter.

5.4.5 Deleting

Command	What will be deleted
x	The current character
<i>nx</i>	<i>n</i> consecutive characters starting from the current character
dw	The current word
D	From the current character to the end of the line
dd	The entire current line
dtc	The current line through the next occurrence of character <i>c</i>
:<i>i</i>d	Line number <i>i</i>
:<i>i</i>,<i>j</i>d	Line number <i>i</i> until (and including) line number <i>j</i>
:1,.d	The first line of the file through (and including) the current line
:.,\$d	The current line through (and including) the last line of the file

In the last two commands the dot (.) refers to the current line. The \$ in the last command refers to the last line.

5.4.6 Moving and copying

Command	Explanation
<code>:i mj</code>	Move line <i>i</i> , insert after line <i>j</i>
<code>:i,jmk</code>	Move lines number <i>i</i> through <i>j</i> (inclusive), insert after line <i>k</i>
<code>:icoj</code>	Copy line number <i>i</i> , insert after line <i>j</i>
<code>:i,jcok</code>	Copy lines number <i>i</i> through <i>j</i> (inclusive), insert after line <i>k</i>

5.4.7 Miscellaneous

Command	Effect
<code>:j</code>	Go to line <i>j</i> (cursor at the beginning of this line)
<code>u</code>	Undo the most recent change in the current line
<code>U</code>	Restore the current line to the state when it was first visited
<code>xp</code>	Transpose the current character with the one next to it
<code>ddp</code>	Exchange the current line with the next line
<code>~</code>	Toggle the case of the current character
<code>n~</code>	Same as above, but for <i>n</i> consecutive characters
<code>:help ccc</code>	Get information about command <i>ccc</i>

The `xp` (transpose) command corrects a lot of typing errors by exchanging the order of two consecutive characters.

In case-sensitive programming languages such as Java and C many mistakes pertain to a wrong case. The `~` (tilde) command is convenient to correct this.

5.4.8 Saving and closing file

Command	Effect
<code>:w</code>	Write to file (save it)
<code>:q</code>	Quit the vi session
<code>:q!</code>	Quit without saving
<code>:wq</code>	Save first, then quit

5.5 To probe further

For more information about the vi editor, you can study the tutorial that is available on your system. Copy the file `vimtutor` from the `/usr/bin` directory (or from some other directory, depending on your installation) to your home directory. Then execute `vimtutor` from your home directory. In addition to the explanations, there are also exercises to practice the commands.

You can also refer to *Learning the Vi Editor*, by Linda Lamb, O'Reilly & Associates Inc.

5.6 An editor for the CMS aficionados

We briefly discuss an editor called **the** here. Developers coming from the VM/CMS platform will find **the** (the Hessling editor) quite familiar since **the** is developed to mimic XEDIT, and incorporates features of Manfield Software's Kedit. For more information check:

<http://hessling-editor.sourceforge.net/>

Create a file called `.therc` in your home directory containing these lines:

```
SET COMPAT XEDIT XEDIT XEDIT
SET CASE MIXED IGNORE
SET CURLINE 3
SET CMDLINE TOP
SET SCALE ON 1
SET NUM ON
```

A screen sample using this `.therc` is shown Example 5-2 on page 73. Don't forget you are working with Linux, not with CMS!

Example 5-2 Sample screen of the

/home/somebody/pgm03.java

====>

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+
00000 *** Top of File ***
00001 /** Show makes a component visible; this method became deprecated
00002 public void show() {
00003     setVisible(true)
00004 }
00005
00006 /** An applet must have a public no-argument constructor.
00007  * @throws java.lang.IllegalArgumentException on Sundays.
00008  */
00009 public JavadocDemo() {
00010 if (new java.util.Date().getDay() == 0 {
00011     throw new IllegalAgumentException("Never on a Sunday");
```

THE3.0

Files=1

Width=512

CR

The Jakarta project

Jakarta is a project hosted by the Apache Software Foundation. The Jakarta project currently consists of twenty-two subprojects, all written in Java. The projects are categorized into the following groups:

- ▶ Libraries, Tools, and APIs
- ▶ Frameworks and Engines
- ▶ Server Applications

For a complete description of all the Jakarta subprojects, refer to the Jakarta home page:

<http://jakarta.apache.org/>

In this chapter, we introduce:

- ▶ Jakarta Tomcat application server
- ▶ Jakarta Ant, a make replacement
- ▶ Jakarta Log4j
- ▶ Jakarta Taglibs
- ▶ Jakarta Struts

6.1 The Tomcat application server

Tomcat is the designated reference implementation for the Java servlet and JSP standard. It can be used as a lightweight stand-alone server for testing servlets and JSP pages, or it can be integrated into the Apache Web server.

For the authoritative documentation, see:

<http://jakarta.apache.org/tomcat/tomcat-4.0-doc/index.html>

6.1.1 Obtaining Tomcat

The most convenient method to install Tomcat is to use RPM packages. Other Jakarta components are required in addition to Tomcat. For this redbook, we used the most current stable release (V4.0.3) available at:

<http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.3/rpms/>

A complete list of all required components is summarized in Table 6-1.

Table 6-1 Components required for Tomcat installation

File name	Description
regex-1.2.1.noarch.rpm	Java regular expression package
servletapi4.4.03-1.noarch.rpm	Servlet container
xerces-j-1.4.4-2.noarch.rpm	Java XML parser
tomcat4-4.0.3-1.noarch.rpm	Tomcat server
tomcat4-webapps-4.0.3-1.noarch.rpm	Sample applications for Tomcat

6.1.2 Installing Tomcat

To install Tomcat, use `rpm` to install the downloaded packages, then create the `userid` and `group` under which the server will run (typically named `tomcat4`). These steps are illustrated in Example 6-1.

Example 6-1 Tomcat installation

```
$ rpm -ivh regex-1.2.1.noarch.rpm \  
    servletapi4.4.03-1.noarch.rpm \  
    xerces-j-1.4.4-2.noarch.rpm \  
    tomcat4-4.0.3-1.noarch.rpm \  
    tomcat4-webapps-4.0.3-1.noarch.rpm  
$ groupadd tomcat4  
$ useradd -c 'Tomcat4 user' -d /var/tomcat4 -g tomcat4 \  
    -s /bin/false -p password tomcat4
```


Starting and stopping the server

Next, modify Tomcat's startup configuration file `/etc/tomcat4/conf/tomcat4.conf`, assigning the `JAVA_HOME` variable to the IBM J2SDK home directory and the `TOMCAT_USER` to the previously created userid:

```
.  
. # Where your Java installation lives  
JAVA_HOME=/opt/IBMJava2-s390-131/  
. # What user should run tomcat  
TOMCAT_USER=tomcat4
```

The Tomcat server startup and shutdown script is `/etc/rc.d/init.d/tomcat4`. To start Tomcat issue the command:

```
/etc/rc.d/init.d/tomcat4 start
```

To stop Tomcat issue the command:

```
/etc/rc.d/init.d/tomcat4 stop
```

Note: Tomcat configuration file

The Tomcat startup configuration file (`/etc/tomcat4/conf/tomcat4.conf`) is sourced by the server startup script (`/etc/rc.d/init.d/tomcat4`). Variable assignments in the startup configuration file equate to environment variable assignments seen by the Tomcat server on startup. Another important variable specified in the startup configuration file is `CATALINA_HOME`, the Tomcat server root directory.

Note: Tomcat Stop

If you stop Tomcat and it is already stopped, you see the following Java exception:

```
Catalina.stop: java.net.ConnectException: Connection refused  
java.net.ConnectException: Connection refused  
    at java.lang.Integer.equals(Integer.java(Compiled Code))  
    at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:345)  
    ...
```

Use the command `ps -ef | grep tomcat` to see if Tomcat is still alive. If not, find the main Tomcat process using `ps -Heff | less` and kill this process. The dependent processes should also stop.

Verifying the installation

By default, Tomcat will listen on port 8180 for incoming requests. To verify the installation has succeeded, in a browser attempt to access URL:

`http://localhost:8180/`

This should bring up the home page of the Web application that comes with Tomcat. Navigate to the Servlet examples page and execute some of the example servlets to ensure the container is working correctly.

6.1.3 Configuring the Tomcat server

The Tomcat server is configured using an XML configuration file found in the `$CATALINA_HOME/conf` directory. The server configuration file defaults to `/var/tomcat4/conf/server.xml`. The complete server configuration reference can be found at:

<http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config/index.html>

In brief, the Tomcat server port defaults to 8180 to avoid conflicts with an existing Tomcat 3 server (which by default uses port 8080), and to permit non-privileged users to start the server (which implies a port greater than 1024). A summary of the server directory structure is given in Table 6-2.

Table 6-2 Tomcat server directory structure

Directory	Contains
<code>\$CATALINA_HOME/bin</code>	Scripts for startup, shutdown etc.
<code>\$CATALINA_HOME/common/lib</code>	Jar files that are shared between Tomcat components
<code>\$CATALINA_HOME/conf</code>	Tomcat configuration files: <code>server.xml</code> and <code>web.xml</code>
<code>\$CATALINA_HOME/lib</code>	Jar files for the servlet engine
<code>\$CATALINA_HOME/log</code>	Tomcat log files
<code>\$CATALINA_HOME/server</code>	Jar files
<code>\$CATALINA_HOME/temp</code>	Miscellaneous temporary files
<code>\$CATALINA_HOME/webapps</code>	Tomcat Web application directory
<code>\$CATALINA_HOME/work</code>	Servlets generated from JSPs

6.1.4 Deploying an application under Tomcat

The easiest way to deploy a Web application is to first package it as WAR file. Copy the .war file to the directory \$CATALINA_HOME/webapps. In compliance with the servlet API 2.3, Tomcat will create all the necessary files and subdirectories from the .war file so that everything is ready when the server is restarted. This feature is referred to as *auto-deploy*. Once restarted, the application can be accessed from URL:

```
http://localhost:8180/appl-name
```

where *appl-name* is the application name specified in the application Web descriptor file (\$CATALINA_HOME/webapps/*appl-name*/WEB-INF/web.xml).

6.1.5 Tomcat application manager

Tomcat provides a special Web application to provide application deployment functions - the *application manager*. Using the manager, it is possible to list, deploy, restart, and stop applications running under the server. For security reasons, the application manager is disabled by default. To enable it, add the following line to the Tomcat \$CATALINA_HOME/conf/tomcat-users.xml configuration file:

```
<user name="userid" password="passwd" roles="manager"/>
```

where *userid* is the user under which Tomcat runs, and *passwd* is the corresponding login password.

After restarting Tomcat, the application manager be accessed from an HTTP agent using requests of the form:

```
http://tomcat-server/manager/command?parameters
```

where

<i>tomcat-server</i>	is the Tomcat server (hostname and port number)
<i>command</i>	is the manager command
<i>parameters</i>	are parameters to be passed to <i>command</i>

Commands provide for:

- ▶ Listing currently deployed applications
- ▶ Deployment of new applications
- ▶ Reloading existing applications
- ▶ Stopping deployed applications

For a complete description, consult the manager documentation at:

<http://jakarta.apache.org/tomcat/tomcat-4.0-doc/manager-howto.html>

6.2 Ant

Ant is a Java-based replacement for the UNIX make utility (see “Automating the build process” on page 14). In the same way make operates on an input configuration file, Ant also takes an input configuration file (named `build.xml`) to control what parts of a project are to be built, and how.

6.2.1 Installing Ant

The current release of Ant (v1.4.1) can be obtained in binary form at:

<http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/bin/>

Download both the base install file and the optional task file:

```
jakarta-ant-1.4.1-bin.tar.gz
jakarta-ant-1.4.1-optional.jar
```

Unpack the distribution into the desired directory (for instance `/usr/local/ant`), define that directory in the `ANT_HOME` environment variable, and ensure the `$ANT_HOME/bin` directory is added to `$PATH`.

Important: The Ant optional package is required for our build process (see “Packaging applications for deployment” on page 251) because we use the task `javah`.

6.2.2 Using Ant

Ant utilizes concepts similar to make: targets may specify prerequisites, tasks correspond to makefile rules and define the steps to build targets, properties correspond to makefile variables. A number of predefined properties and tasks are available. Ant attempts to simplify the process of building an application: it operates on an XML file, and its built-in rules are designed to be platform independent.

To invoke ant, simply issue the command **ant**. A sample `build.xml` file is shown in Example 6-2.

Example 6-2 Sample Ant build.xml file

```
<?xml version="1.0"?>
<project name="sg246807" default="compile" basedir=".">
  <property name="src" value="src" />
  <property name="build" value="build" />
  <property file="build.properties" />
```

1
2
3

```

<target name="init">
    <tstamp/>
    <mkdir dir="${build}" />
</target>

<target name="compile" depends="init" description="compile phase" />
    <javac srcdir="${src}" destdir="${build}" />
</target>
</project>

```

1. The project statement defines a default target (compile) and sets all directory paths relative to the current directory.
2. Variable names are assigned values using the property statement. All system properties and a number of Ant built-in properties are also available.
3. Additional property values may be defined in an external file and imported.
4. Target `init` defines initialization tasks.
5. The `tstamp` task sets a timestamp on the created directory to ensure the task is executed only if the target is out-of-date.
6. The Ant-defined `mkdir` task creates the build directory.
7. Target `compile` defines the compilation phase. It depends upon the `init` task, and provides a description attribute that will appear on output when issuing the command:

```
ant -projecthelp
```

Ant has over 70 built-in task definitions; see the Ant reference manual for complete details at:

<http://jakarta.apache.org/ant/manual/index.html>

6.3 Log4J

Log4j is a Java package designed to provide runtime logging with little performance overhead. Using Log4j, you can build logging into an application and not be overly concerned with incurring a performance penalty when the application goes into production, since you can simply disable logging using the Log4j runtime configuration.

6.3.1 Installing Log4j

1. Download the latest release of Log4j. We used `jakarta-log4j-1.2rc1.zip`:

<http://jakarta.apache.org/log4j/docs/download.html>

2. Uncompress the distribution with:

```
unzip jakarta-log4j-1.2rc1.zip
```

and add the file `dist/lib/log4j-1.2rc1.jar` to the `WEB-INF/lib` directory.

For information on using Log4j in your application, refer to “Logging using Log4j” on page 176.

6.4 Taglibs

Tag libraries, or *taglibs*, are custom tags which may be included in JavaServer Page (JSP) files. Taglibs alleviate the need to include custom scripting code in JSP files. They encourage reuse of customized presentation features because tags are referenced inside JSP files—not simply copied as blocks of script. There are several taglibs packages available from the Jakarta Taglibs site:

<http://jakarta.apache.org/taglibs/index.html>

These packages are grouped by functionality and are described in the contents.

6.4.1 Installing taglibs

Taglibs are distributed with:

- ▶ A tag library descriptor (TLD) file
- ▶ A Java implementation (jar) file

Tag library descriptors

Taglibs are declared in tag library descriptors (TLD), and packaged as `.tld` files. A TLD defines a tag library and its respective tags. Descriptors contain metadata about tags, such as tag name and parameters. The elements in the TLD document are used by the JSP page compiler to access the tag Java components.

Installing tag Java classes

The tag Java classes must be available to the Web application context. Tag Java classes are packaged as a JAR and deployed in the Web application `WEB-INF/lib` directory.

6.4.2 Configuring taglibs

Using taglibs as part of a Web application requires some setup before tags can be used in JSP pages. Taglibs must be visible to the Web application context to use JSP tags.

Declaring taglibs in the Web application descriptor

A Web application descriptor points to existing taglibs that will be used by JSP pages. Taglibs are referenced and mapped in a Web application descriptor, WEB-INF/web.xml. Taglib URIs are mapped to a local or remote location which contains the tag library descriptor. In Example 6-3, the Struts HTML taglib is mapped to a local Web application path.

Example 6-3 Struts taglib definition [WEB-INF/web.xml]

```
<taglib>
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
```

Using tags in JSP pages

To use JSP tags, the taglib directive must be included in the JSP page. A JSP page can use tags from more than one taglib, but each taglib used must be declared in a directive. Example 6-4 illustrates taglib usage in a JSP page.

Example 6-4 Using taglib directives in a JSP page

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
<head>
  <title>SG246807 - Taglib Usage</title>
</head>
<body>
  <h1>Using Taglibs</h1>
  <ul>
    <li>
      <html:link page="/useradmin/addUser.jsp">
        Follow this link
      </html:link>
    </li>
  </ul>
</body>
</html>
```

1. Taglib references are defined to the JSP engine using the taglib directive. Two attribute definitions are required:
 - a. uri identifies the application-relative location of the taglib TLD file.
 - b. prefix specifies a unique identifier associated with tags specific to the TLD. (The prefix allows multiple taglibs to be included in a single JSP page.)

2. The `link` tag is used to create an HTML hyperlink to the URL `/useradmin/addUser.jsp`. The `html` prefix refers to the previous `taglib` directive, the `link` tag is defined in the `struts-html.tld` descriptor, and the `page` attribute is a required parameter to the `link` tag.

On compilation, the JSP engine will syntax-check `taglib` usage based on the TLD specification.

6.5 Struts

Struts is a framework based on the Model-View-Controller (MVC) pattern utilizing JSP, taglibs, and Java servlet technology. In brief, the model corresponds to an application's business logic; the view corresponds to the application presentation (typically implemented using JSP); and the controller corresponds to the Struts controller itself. Taglibs and JavaBean technology form the basis of the Struts framework. Struts taglibs are used to standardize HTML form creation and validation, provide simple access to JavaBeans in JSP pages, and enable iterative and condition logic inside JSP pages.

6.5.1 Struts components

We begin by defining some Struts components.

ActionServlet	The Struts controller, a single instance of the <code>org.apache.struts.action.ActionServlet</code> class which controls Struts operation. Applications typically only configure the <code>ActionServlet</code> using the <code>WEB-INF/web.xml</code> file.
Action	An instance of the <code>org.apache.struts.action.Action</code> class that acts on HTTP requests. Applications derive from the <code>Action</code> class to act as request handlers implementing application-specific business logic in the <code>perform</code> method.
ActionMapping	The <code>org.apache.struts.action.ActionMapping</code> class which maps HTTP requests to the <code>Action</code> class instance intended to handle that request. Applications generally specify this mapping in a configuration file (<code>WEB-INF/struts-config.xml</code>).
ActionForward	The <code>org.apache.struts.action.ActionForward</code> class which directs the <code>ActionServlet</code> to forward a request to another <code>Action</code> instance or JSP servlet. <code>ActionForwards</code> are typically returned by <code>Action</code> instances request handlers (to allow <code>Action</code> handlers to be stacked, and JSP servlets to be conditionally executed).

ActionForm The `org.apache.struts.action.ActionForm` JavaBean class associated with `ActionMapping` instances. `ActionForm` instances are initialized by `ActionServlet` from values supplied on HTTP form submission and passed to `Action` request handlers as a parameter to the `perform` method. Applications may derive from `ActionForm` to implement form validation (via the `validate` method).

Figure 6-1 illustrates the relationship between these components.

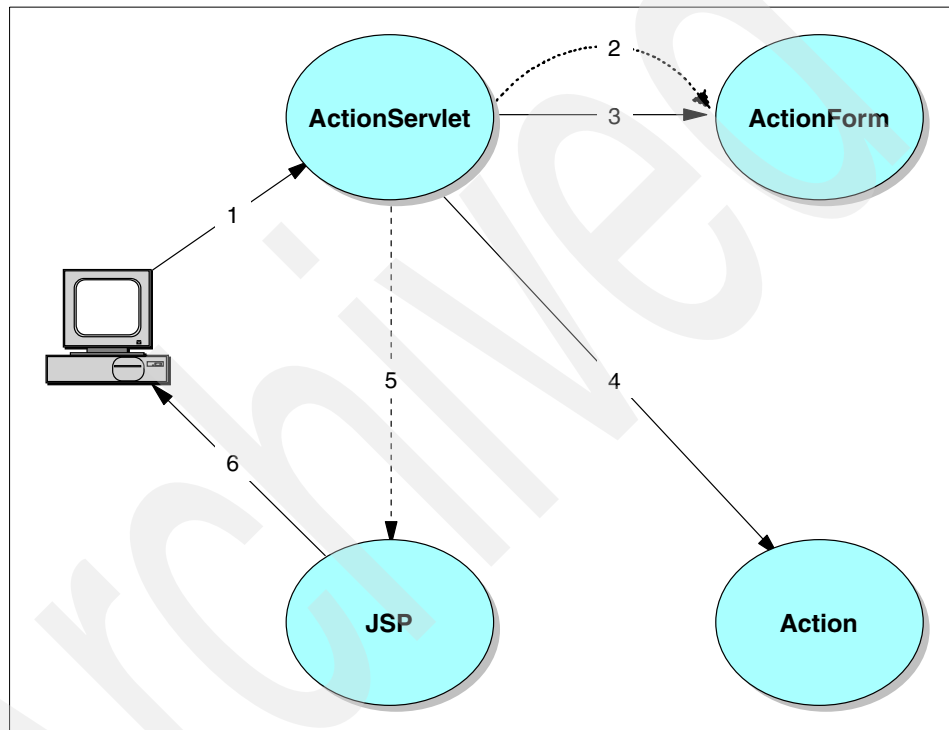


Figure 6-1 Struts framework component interaction

1. An incoming HTTP POST request (generated by an HTML form submittal) is intercepted by the `ActionServlet` instance.
2. An `ActionMapping` associates the request to a specific `ActionForm` instance. The `ActionServlet` instance instantiates an `ActionForm` JavaBean instance based on that mapping. The HTML `<form>` input parameters correspond to data members of the `ActionForm` instance (form parameters are mapped to `ActionForm` instance data by name).

Note: The `ActionForm` instance is instantiated on processing the first HTTP request. `ActionServlet` will reuse this instance on subsequent requests.

3. The `ActionServlet` invokes the `validate` method of `ActionForm`. Any validation errors are returned to the `ActionServlet`.
4. If no validation errors are reported, the `ActionServlet` then invokes the `perform` method of the `Action` instance corresponding to this request (passing the `ActionForm` as a parameter). The `ActionMapping` association determines which `Action` instance to invoke. Business logic in the `Action` instance is executed in the `perform` method. On completion, the `Action` returns an `ActionForward` to the `ActionServlet` (indicating where control should pass next).

Important: Only a single `Action` instance exists per servlet container. It is crucial to ensure the `Action` class implementation be thread-safe.

5. The `ActionServlet` passes the request to the next phase (based on the returned `ActionForward`). In this case, forwarding is directed to a JSP page, but forwarding to another `Action` instance is also possible (stacked handlers). Because forwarding is controlled by the `ActionForward` returned to the `ActionServlet`, the `Action` instance can maintain runtime control over logic flow.
6. An HTTP response is generated and returned to the client.

Struts taglibs

Custom taglibs form an integral part of the Struts framework. Struts taglibs are grouped by functionality:

<code>struts-html</code>	Provides tags that render to HTML elements. These tags are typically used to assist in mapping HTML forms to an <code>ActionForm</code> <code>JavaBean</code> instance.
<code>struts-bean</code>	Provides tags to assist in creating and accessing <code>JavaBeans</code> in JSP pages.
<code>struts-logic</code>	Provides tags to conditionally render HTML elements based on the state of <code>JavaBeans</code> . Iteration logic is also provided.
<code>struts-template</code>	Provides a mechanism to render HTML based on templates (useful for HTML pages that share a common layout structure).

Each of these taglibs is distributed with its own TLD file. Tag implementation is provided in a common jar file (`struts.jar`). The complete Struts taglib reference can be found in the Struts documentation page at:

<http://jakarta.apache.org/struts/doc-1.0.2/>

See the *Taglib Documentation* section.

6.5.2 Installing Struts

Struts can be obtained from the Struts download page:

<http://jakarta.apache.org/builds/jakarta-struts/release/v1.0.2/>

1. Unpack the distribution to a convenient directory (for example, the `/usr/local/struts` directory).
2. Copy the distribution `lib/struts.jar` file to your application `WEB-INF/lib` directory.
3. Copy the distribution taglib descriptor files (`lib/*.tld`) to your application `WEB-INF/lib` directory.

Some example applications are included in the distribution (packaged as WAR files found in the `webapps` directory). We demonstrate how to use Struts in Chapter 13, “Using the Struts framework” on page 167.

Running Linux applications in a zSeries environment

In this chapter we discuss some aspects of running Linux programs on zSeries architecture.

We focus on:

- ▶ zSeries and s/390-specific features
- ▶ Stack layout and function call conventions
- ▶ Debugging with the gdb tool
- ▶ Optimization issues
- ▶ Linux signals

7.1 Architecture consideration

In this section we consider some of the specific features of the zSeries architecture, particularly the differences in comparison to the i386 platform.

7.1.1 Bits and bytes

For the 16-bit (halfword), 32-bit (word), 64-bit (double word) data types, zSeries architecture puts the most significant byte first. This type of byte ordering is often called *big-endian*.

Table 7-1 shows how C data types are implemented in Linux on zSeries.

Table 7-1 ANSI C data types

Type	ANSI C	Size and alignment in bytes
byte character	char unsigned char	1
short (halfword)	short unsigned short	2
integer (word)	int unsigned int enum long unsigned long	4
doubleword	long long unsigned long long	8
pointer	(type *) type (*) ()	4
single precision	float	4
double precision	double	8
extended precision	long double	16

The alignments for these data types is the same as their respective sizes. You should consider this when composing C structures or unions.

Structures

Example 7-1 shows some C structures implemented on zSeries. The structures are of the following sizes:

- struct one is 24 bytes
- struct two is 16 bytes
- struct three is 16 bytes

Example 7-1 Data alignment in C structures

```
struct one{
    short s1;
    long long l1;
    short s2;
};
```

```
struct two{
    long long l1;
    short s2;
    short s1;
}two;
```

```
struct three{
    long long l1;
    char c1;
}three;
```

Unions

When defining a union, its size is the size of its longest component plus a proper alignment.

The order of bits is big-endian, too. Let us have a look at Example 7-2.

Example 7-2 Structure with bit fields

```
union u{
    struct four{
        unsigned char i:1;
        unsigned char j:2;
        unsigned char k:2;
    }four;
    unsigned char all;
}u;
```

When the following two lines are executed:

```
u.all=0;
u.four.i=1;
```

the value of `u.all` is 128.

Maintaining portability

Different architectures and different compilers may have their own approach to the composition of a physical data representation. We suggest creating data structure with types like `int16_t` defined in `sys/types.h` and checking the final layout when sending arrays of structures is over the network.

Before you start to write your own assembler routines performing some operations on bits, you should consult files in the `/usr/include/asm` directory—especially `asm/bitops.h`. You may find some of the routines already written. The advantage is obvious: these routines are likely to be present on the other platforms, so your program has a better chance to be portable.

Note: You can check for the existence of `__platform__` macros to discover the platform you work with. On zSeries, the `__s390__` flag is predefined.

7.1.2 Virtual address space

Processes are executed in the user mode (program state) while the kernel runs in supervisor mode. A supervisor call (`svc`) is used to pass control to a different mode. Virtual memory is addressed in 31-bit mode.

The most significant bit is always set to 0. Figure 7-1 shows the layout of the four process segments:

- ▶ Executable code
- ▶ Heap
- ▶ Dynamic segments (`mmap`)
- ▶ Stack

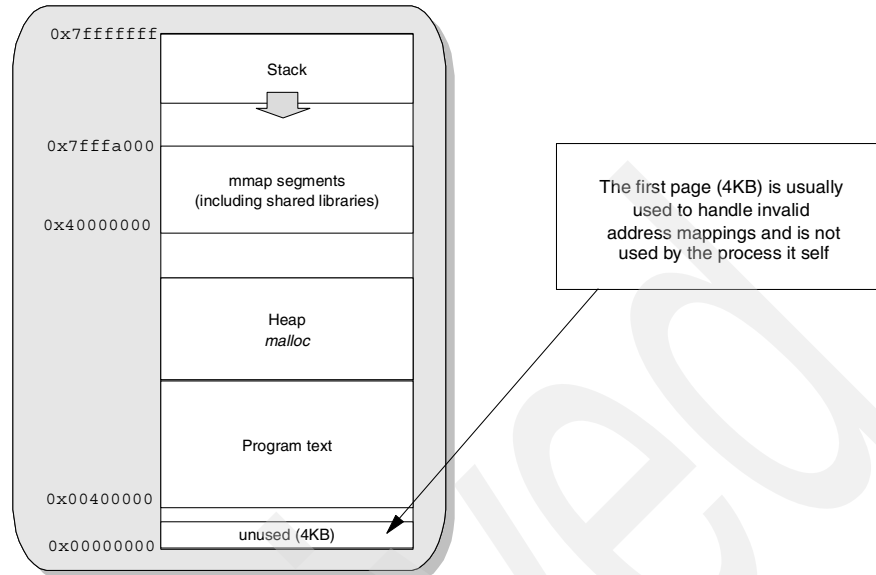


Figure 7-1 Virtual memory layout in Linux on zSeries

A process can also use `setrlimit` to set its own maximum stack size or memory limit. For details, see the man pages for `setrlimit` system calls and `ulimit` bash commands.

Functions that map data into the process virtual memory are all permitted to specify the address where the mapping is to occur. If you use this parameter (for example, on `mmap` or `shmat` function calls), the address you specify must be proper for zSeries architecture. The most significant bit must be set to 0 and the address must be valid.

7.1.3 Function calling convention

When a C function is called (unless in-lined due to optimization), the compiler will save local function-local data on the stack. Some function parameters are passed in registers, others are placed on stack. Register usage is summarized in Table 7-2.

Table 7-2 Register usage in Linux on zSeries

Register	Usage	Saved
R0,R1	General purpose	No
R2	First argument and return value	No
R3	Second argument and second word of return value if necessary	No
R4,R5	Next arguments	No
R6	Next arguments	Yes
R7-R9	Local variables	Yes
R10	static-chain (or a local variable)	Yes
R11	frame-pointer (or a local variable)	Yes
R12	GOT pointer (or a local variable)	Yes
R13	Base-pointer the literal pool which stores constants used in a function	Yes
R14	Return address	No
R15	Stack pointer	Yes
F0,F2	Parameters	No
F4,F6	General purpose	Yes
F1,F3,F5,F7-F15	General purpose	No

If there are more than five arguments, subsequent ones are passed on a stack. Moreover, long long values are passed in two consecutive registers. Double word parameters are never split between the R6 register and stack (in this case, the parameter is placed on stack).

Figure 7-2 presents the stack layout for Linux on zSeries.

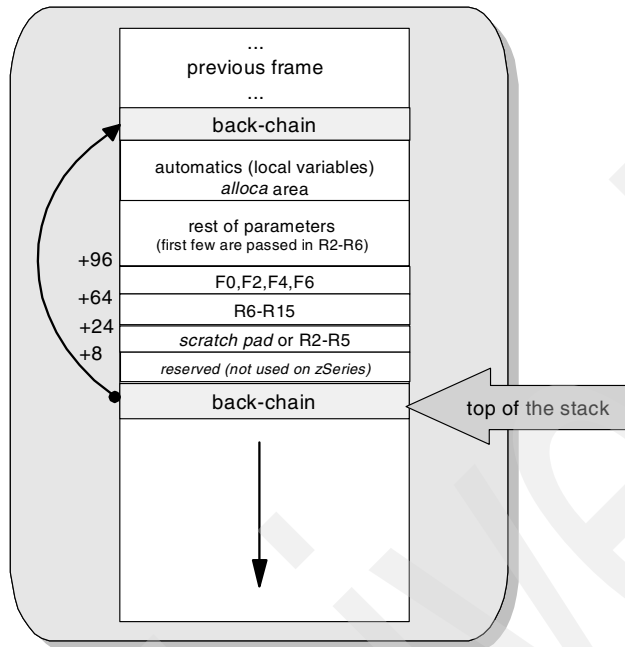


Figure 7-2 Frame layout for Linux on zSeries

Unless the optimization option is in use, a function epilog and prolog are generated as shown in Example 7-3. The code presented here was produced by gcc for a function which takes one parameter (four-byte word) and passes the return value in four-byte word:

```
int f(int i){
    a=i;
}
```

As shown in Example 7-3, we reserve 96 bytes (4 for an automatic variable and 4 for double-word alignment).

Example 7-3 Prolog and epilog for non-optimized code

```
.globl f
        .type    f,@function

f:
#       leaf function           1
#       automatics             8
#       outgoing args          0
#       need frame pointer     1
#       call alloca            0
#       has varargs            0
#       incoming args (stack)  0
```

```

#      function length      14
#      register live       0110000000010100000000000000000010
      stm    %r11,%r15,44(%r15)
      bras   %r13,.LTN0_0
.LT0_0:
.LC0:
      .long   a
.LTN0_0:
      lr      %r1,%r15
      ahi     %r15,-104
      st      %r1,0(%r15)
      lr      %r11,%r15
      st      %r2,96(%r11)
      l       %r1,.LC0-.LT0_0(%r13)
      mvc     0(4,%r1),96(%r11)
.L2:
      l       %r4,160(%r11)
      lm      %r11,%r15,148(%r11)
      br      %r4

```

7.2 When things go wrong

A debugger is a tool that allows a developer to step through code to locate problems. In the following section we describe gdb, one of several such tools available for Linux.

7.2.1 Debugging with gdb

Before you start debugging, you need to include some extra information when compiling your program. This data is stored in the object file; it describes the data type of each variable or function, and the correspondence between source line numbers and addresses in the executable code.

To include debugging information, specify the **-g** option for every module you compile.

Let's look at a short session with gdb. Example 7-4 shows `faulty.c`, the program we would like to debug.

Compile the code with this command:

```
gcc -g -o faulty faulty.c
```

The first time you run it, you should see the following output:

```
$faulty
```

```
i = 00
i = 01
i = 02
i = 03
i = 04
i = 05
Segmentation fault
```

The program was stopped by the system since it tried to access an invalid area in memory. It is much easier to figure out what has happen when we have a snapshot of memory at the time of termination.

Example 7-4 faulty.c

```
#include<stdio.h>

int a[10];

buggy(int arg){

    int i;
    int *p1=0;
    int *p2=a;

    i=13;
    *p1=i; /* 'p' in not initialized */

    printf("Am I still alive ?\n");
}

loop(int arg){
    int i;

    for(i=0; i<arg; i++){

        printf("i = %02d\n", i);
        a[i]=i;

        if( i % 10 == 5 )
            buggy(i);

    }
}

main(int argc, char *argv[]){
    loop(10);
}
```

By default, core dumps are not stored in the file system:

```
$ulimit -a
data seg size (kbytes)      unlimited
file size (blocks)         unlimited
max locked memory (kbytes)  unlimited
max memory size (kbytes)   unlimited
open files                 1024
pipe size (512 bytes)      8
stack size (kbytes)        unlimited
cpu time (seconds)         unlimited
max user processes         256
virtual memory (kbytes)    unlimited
```

We can change this using ulimit with the **-c** option in shell:

```
$ulimit -c unlimited
```

Now we can run our program once again.

```
$faulty
i = 00
...
i = 05
Segmentation fault (core dumped)
```

Since we have the contents of memory stored in the core file, we can ask gdb to work with both the program code and the snapshot.

```
$gdb buggy core
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "s390-suse-linux"...
Core was generated by './faulty'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x804842a in faulty (arg=5) at faulty.c:13
13      *p1=i;
(gdb)
```

If the program was compiled with the `-g` option, gdb analyzes the core dump file and can immediately point out where the execution was interrupted: line 13 from the file `faulty.c`.

We can look at how the function containing this line was called.

```
(gdb) info stack
#0  0x804842a in buggy (arg=5) at buggy01.c:13
#1  0x80484c9 in loop (arg=10) at buggy01.c:29
#2  0x80484f0 in main (argc=1, argv=0xbffff674) at buggy01.c:35
#3  0x40041c6f in __libc_start_main () from /lib/libc.so.6
```

We can also examine variable values as if the program was still being executed:

```
(gdb) p i
$1 = 13
```

If you need access to a variable from a different scope, use the operator `::`, as follows:

```
(gdb) p loop::i
$2 = 5
```

Starting your program

Type `run arg1 arg2 ...` to start a program under gdb. Note that the working directory and standard input and output are inherited from gdb.

attach <i>pid</i>	Attach to a running process that was started outside gdb.
detach	Release the program from gdb control.
kill	Stop the child process in which your program is running under gdb.

Linux threads are also supported by gdb. You can be notified when a new thread begins execution and use thread-specific breakpoints. Use the following commands:

thread <i>tid</i>	Switch among threads.
info threads	Inquire about existing threads.
thread apply	Apply a command to a list of threads.

Breakpoints

Breakpoints are used to inform the debugger where it should stop execution of a program.

break <i>line</i>	Set a breakpoint at <i>line</i> in the current file.
break <i>file:line</i>	Set a breakpoint at <i>line</i> in <i>file</i> .

break <i>function</i>	Set a breakpoint at the start of <i>function</i> in current file.
break <i>file:function</i>	Set a breakpoint at the start of <i>function</i> in <i>file</i> .
break	Set a breakpoint at the next instruction to be executed in the selected stack frame. This is useful when debugging loops and functions.
break ... if <i>expr</i>	Set a breakpoint with condition <i>expr</i> . The expression <i>expr</i> is evaluated each time the breakpoint is reached, and the execution stops only if the expression is true (non-zero).
ignore <i>bnum count</i>	Ignore <i>count</i> passes through breakpoint number <i>bnum</i> .
info break	Shows information on breakpoints.
watch <i>expr</i>	Watch an expression. The gdb will break when <i>expr</i> is changed.

Note: The gdb can only watch the value of an expression in a single thread.

Source files

gdb maintains the list of directories to search for source files, which is called the *source path*. By default this list contains only the compilation directory (*cd i r*) and the working directory (*cwd*).

To add other directories, use the *directory* command.

dir <i>dirname ...</i>	Add directory <i>dirname</i> to the front of the source path. Multiple entries are separated by a colon (:).
directory	Reset the source path to an empty one.
show directories	Print the source path: show which directories it contains.

You can list the source code with the following commands:

list <i>linenum</i>	Print lines centered around line number <i>linenum</i> in the current source file.
list <i>function</i>	Print lines centered around the beginning of <i>function</i> in the current file.
list	Print more lines.

Printing data

print <i>expr</i>	Prints value of an <i>expr</i> .
--------------------------	----------------------------------

Useful operators for this command are:

<code>::</code>	Allows you to specify a variable in terms of the file or function where it is defined.
<code>{type} addr</code>	Evaluates to an object of <i>type</i> stored at address <i>addr</i> in memory (<i>addr</i> may be any expression whose value is an integer or pointer).
<code>display expr</code>	Expression <i>expr</i> is printed each time the program stops.

7.2.2 Tracing system calls

The `strace` tool intercepts and records system calls invoked by a process. In addition, signals received by the process are intercepted. The name of each system call, its arguments, and its return value are printed (to standard error or to the file specified with the `-o` option).

The `ptrace` system call provides a means by which a parent process may observe and control the execution of a child process. The child's core image and registers can be examined and altered. This call is primarily used to implement breakpoint debugging and system call tracing.

<code>-c</code>	Count time, calls, and errors for each system call and report a summary on program exit.
<code>-T</code>	Show the time spent in system calls. This records the time difference between the beginning and the end of each system call.
<code>-p pid</code>	Attach to the process with the process ID <i>pid</i> and begin tracing. The trace can be terminated by issuing an interrupt signal (Ctrl-C).
<code>-u username</code>	Run command with the user ID, group ID, and supplementary groups of <i>username</i> .
<code>-f</code>	Trace child processes.

The `strace` command traces system calls only. You will notice no `printf` or `pthread_create` in `strace` output. The list of system calls is available in the `/usr/include/asm/unistd.h` header file.

You will find this tool very handy when you want to get to know what files are opened by the application or what DLLs are loaded (and from which directory).

Tip: If you want to call a system function directly from your program, use `_syscalln` from `/usr/include/asm/unistd.h`

7.2.3 Debugging under zVM

zVM has its own debugger that can be used to track Linux activities. Refer to *Debugging on Linux for 390* by Denis Joseph Barrow. You can find this comprehensive document in your kernel source directory or at:

<http://linuxvm.org/penguinvm/notes.html>

The zVM reference books are available at:

<http://www.vm.ibm.com/library/>

The zVM debugger is useful for debugging kernel activity. However, you can also use it to trace the system calls. If you know the system call number (see the `/usr/include/asm/unistd.h` header file), turn on tracing with the following `cp` command:

```
#cp trace svc syscall_number
```

To stop tracing:

```
#cp trace end
```

7.2.4 Performance profiling

In addition to option `-g` (which includes debugger information), you can compile your program with support for profiling using the `-gp` option. The `prof` tool allows you to determine the execution time for each part of your program. The profiler that is available in Linux installations is a GNU program called `gprof`. You can find a `gprof` tutorial in the info help system, and on the Internet at:

<http://sources.redhat.com/binutils/docs-2.10/gprof.html>

7.3 Optimizing for performance

In this section we discuss some of the `gcc` compiler options that can improve performance of an application.

7.3.1 General options

The main switch for optimization in the `gcc` compiler is `-On`. The higher the number, the better optimizing. By default, the optimization is turned off (`-O0`). The most frequently used level is `-O2`, which gives (in most cases) the best performance without necessarily enlarging code size or introducing significant changes that may lead to performance degradation. The `-On` option is an abbreviation for more specific switches set with `-ffeature`.

7.3.2 Inline functions and unrolled loops

Inline functions are a convenient programming feature. You can decompose your application into small, easy-to-understand fragments without worrying about the overhead caused by epilog and prolog code (the compiler expands inlined functions directly where the function call is made). To use inline functions, you must specify the `-finline-functions` option in addition to the `-O2` option when compiling.

Note: Be advised that this mechanism does not work for library functions. If you want some of your library functions to be compiled into the code on every call, you should define them with the `inline` attribute in the library header file explicitly.

The `-funroll-loops` allows the optimizer to repeat the loop statements one after the other when the number of iterations can be determined at compile time or run time.

Note: The optimization process (particularly the two options described here) may produce slower code than an unoptimized version. This is often related to the effects optimization has on paging and CPU caching. We found that loops containing function calls often performed better when the `-funroll-loops` option was not specified.

7.3.3 Architecture-dependent options

We now look at architecture-dependent gcc options.

-mno-backchain / -mbackchain

This option eliminates a back-chain:

```
lr      %r1,%r15
...
st      %r1,0(%r15)
```

It saves a few cycles on every function call.

-msmall-exec / -mnosmall-exec

By default, function calls are implemented with BASR (BRANCH AND SAVE) instruction. The `-msmall-exec` option can be used to instruct the compiler that a small executable is to be produced. In this case, the relative jump BRAS (BRANCH RELATIVE AND SAVE) instruction can be used. This instruction uses

16-bit offset (in halfwords). This limits branches to locations within a 64 KB range in both directions.

-mhard-float / -msoft-float

Instructs the compiler to use hardware (`-mhard-float`) or software (`-msoft-float`) floating-point instructions and registers for floating-point operations. When option `-msoft-float` is specified, functions in `libgcc.a` will be used to perform floating-point operations. The `-mhard-float` is the default.

-m31 / -m64

When `-m31` is specified, `gcc` generates code compliant to the Linux for S/390 API. The `-m64` option generates code compliant to the Linux for zSeries API (allowing the compiler to generate 64-bit instructions). For s390 targets the default is `-m31`, s390x targets default to `-m64`.

-mmvcl e / -mno-mvcl e

Instructs the compiler to use the `mvcl e` (`-mmvcl e`) or to `mvcl` (`-mno-mvcl e`) instruction to perform block moves (`-mno-mvcl e` is the default).

7.3.4 String operations

On the i386 platform, `gcc` uses inline replacements for string functions like `strcpy` or `strlen` when optimization is enabled. For the zSeries platform, you must enable string optimization using the `-D__USE_STRING_INLINES` compiler option accompanied by `-O2` option.

7.3.5 Sources of information

An excellent reference - *Optimizing with gcc on Linux for S/390*, by Dr. Eberhard Pasch - can be found at

http://www-1.ibm.com/servers/esdd/articles/gcc_opt/index.html

The GCC Documentation pages can be found at

<http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc.html>

7.4 Signals

Signals are a basic method of communication between processes in UNIX. They do not pass any information (except for a signal type), but can trigger an action in a target process.

A signal can be generated by the system or by another process with the `kill` system call. Signals can be intercepted by a process using signal handlers. See the `kill` and `signal` pages in section 2 of the man manual. The complete list of signals is provided by the `signal` man page in section 7:

`man 7 signal`

Important: Since there is no single process ID in multithreaded Linux programs (see 15.2, “The pthreads library” on page 219), each thread has its own PID number and the kernel delivers the signal to that thread.

For more information about how the application can handle signals, see:

- ▶ *The Linux Programmer's Guide*
<http://www.linuxhq.com/guides/LPG/>
- ▶ The article *A Look at the Signal API* by Erik Troan, available at:
http://www.linux-mag.com/2000-01/compile_01.html

7.4.1 Linux signals and zSeries exceptions

The zSeries architecture raises an exception when a program tries to execute an illegal operation. When the exception occurs, the program context is saved and execution flows to the address pointed to by a vector predefined for the exception.

The operating system handles an exception either by completing the faulty operation or by delivering a signal to the process. The correspondence between exceptions and signals is defined in a kernel source file (`trap_init` function) found in the `linux/arch/s390/kernel/traps.c` file in the kernel source.

Although Linux implementation of signals covers most of System V and BSD features, there are still some differences.

For more information on zSeries exceptions, see *z/Architecture Principles of Operation*, SA22-7832.

The Eclipse logo, which consists of two overlapping circles, each containing a map of the Earth, is positioned in the upper left corner of the page.

Part 2

Eclipse

In this part we introduce Eclipse, an open source Integrated Development Environment.

Eclipse overview

In this chapter we introduce Eclipse as a tool development platform for Linux on zSeries. We begin our discussion by looking at Eclipse at its highest level. Then, we describe the building blocks that make up Eclipse. Finally, we point you to where you can get started with Eclipse.

8.1 Eclipse Software Developer Kit

Eclipse at its highest level is known as the Eclipse Software Developer Kit (SDK). The Eclipse SDK consists of three parts: the platform, the Java Development Tools (JDT), and the Plug-in Development Environment (PDE).

In short, Eclipse provides a development environment that allows tool developers to build integrated tools within Eclipse. A tool that has been integrated with Eclipse is known as a plug-in. An example of a plug-in is the C/C++ Development Tool (CDT). A plug-in can facilitate further development of applications in a particular environment. In the case of the CDT, the plug-in becomes an IDE for C/C++ applications development.

The next few sections focus on the three components that make up the Eclipse SDK.

8.2 The Eclipse platform

The platform provides the frameworks that allow tool builders to create an Eclipse plug-in. The platform also provides the runtime out of which plug-ins are loaded, integrated, and executed.

The platform itself provides the IDE and framework for the development of plug-ins while maintaining file type independence. For example, C files or HTML files are just files to the platform. They do not promote a particular behavior from it. However, the plug-in itself can teach Eclipse how to behave according to a specific file type under the plug-in environment. Therein lies the value of the Eclipse platform.

The platform can further be divided into components. The rest of this section will describe the components that build the platform.

8.2.1 Ant

Ant is a Java build tool similar to **make** that has been incorporated into Eclipse. For more information about Ant see 6.2, “Ant” on page 80. This component allows users to run Ant scripts on Eclipse resources. Ant works on Eclipse projects by running against the information contained in the project’s xml and build properties files.

8.2.2 Compare

The universal compare facility of Eclipse allows the user to compare multiple resources. It displays the results inside a special editor. The editor allows the user to manipulate changes between resources. In addition, the facility allows the user to revert to previous copies in the local history contained within Eclipse. With this functionality, Eclipse allows an extra level of recovery.

8.2.3 Core

The core component controls the platform runtime configuration. It is responsible for loading and running the platform. In addition, it is responsible for plug-in and resource management.

8.2.4 Debug

The debug component provides a way to facilitate language-independent debug functionality. This is achieved through a language-independent model which provides abstracts of commonly used debug functions in various languages.

8.2.5 Help

The help component is responsible for presenting the online documentation as well as F1 functionality. The online documentation is provided through HTML files and it is displayed using native HTML browsers.

8.2.6 Release Engineering

The Release Engineering (Releng) component focuses on release issues. It provides for naming, coding, and documentation conventions to ensure that the product is delivered as a cohesive product and not a bunch of parts placed together.

8.2.7 Scripting

The scripting component allows Eclipse script support using the Rhino script engine. This engine provides the features of JavaScript, direct scripting of Java, a shell for executing JavaScript scripts, and a compiler to transform JavaScript files into Java classes. The component allows for workbench extensions and plug-in extensions using scripts.

8.2.8 Search

The search component handles the search capability that is found within the workbench. The base component provides text searches and its functionality can be extended by other plug-ins.

8.2.9 Standard Widget Toolkit

The Standard Widget Toolkit (SWT) component provides the windowing system for Eclipse. This windowing system has been developed to allow development of tools with native environment look and feel.

8.2.10 User Interface

The User Interface (UI) component provides a way to build user interfaces with Eclipse. JFace, the workbench, and the standard subcomponent make up the UI component. JFace is designed to work with SWT and includes image, font registry, text, dialog, wizard preference, frameworks, and progress reporting components for long-running tasks. The workbench provides the user interface for Eclipse. The standard subcomponent includes editors and views that allow the user to navigate resources, present outlines and properties, and manage bookmarks and tasks.

8.2.11 Update

The update component provides a mechanism to locate new plug-ins, add new plug-ins to Eclipse, update existing plug-ins, and manage user Eclipse installation configurations.

8.2.12 Version Control Mechanism

The Version Control Mechanism (VCM) component provides a mechanism for team development support. Currently the VCM works with CVS to provide a team development solution for projects developed under Eclipse.

8.2.13 WebDav

The Web distributed authoring and versioning component (WebDav) client allows Eclipse tools access to the WebDav technology. WebDav is a standard set of extension for HTTP 1.1. WebDav adds methods to manage documents in a Web environment.

8.3 The Java Development Toolkit

The Java Development Toolkit (JDT) is a complete Java IDE for Java application development, including Eclipse plug-in development. The JDT provides the Eclipse workbench with a Java perspective. As such, it provides a view complete with editors and other tools inherent to Java application development. We can further view the JDT as the sum of three components: Core, Debug, and UI subcomponents. Each provides its own functionality to the perspective.

8.3.1 JDT Core

The JDT Core component is responsible for the incremental Java compiler. It provides a Java model for navigation, as well as documentation of Java projects and resources. In addition, it provides code assistance plus source code formatting functionality during development.

8.3.2 JDT Debug

The JDT Debug component provides the debugging capability within the Java perspective. It is responsible for providing launch functions in running and debug modes, the attaching to a running JVM, expression evaluation within a stack frame, and dynamic class reloading where supported by the JVM.

8.3.3 JDT UI

The JDT UI provides the user interface for the Java perspective. It arms the user with various project views with which to see the project at different levels and formats. In addition, it provides a Java editor with syntax coloring, checking, and code assist functions, along with wizards to create Java elements like projects, packages, classes, and interfaces.

8.4 The Plug-in Development Environment

In order to create a plug-in, a user needs to specify a plug-in manifest, specify a run time environment, describe required plug-ins, specify extension points, and specify required schema. The Plug-in Development Environment (PDE) is a perspective that allows the user to describe the items required by a plug-in in a fast and easy way through editors, views, and wizards. The PDE itself consists of the Core and UI subcomponents.

8.4.1 PDE Core

The function of the Core component is to facilitate the building and packaging of plug-ins. It uses Ant, in combination with the plug-in manifest and build properties files, to generate scripts to package the plug-in in a format that is ready for distribution.

8.4.2 PDE UI

The UI component aids in the building of plug-ins by providing the user with Eclipse patterns, testing capabilities, trouble shooting capabilities, and packaging functions.

8.5 Getting started with Eclipse

The focal point for Eclipse development and resources is found at:

<http://www.eclipse.org/>

Resources like available code downloads, Frequently Asked Questions (FAQ), and the problem support database are found there. A word of advice: always download the most stable build for a specific distribution. Integration and nightly builds can be fairly unstable.

Installing Eclipse

This chapter describes the installation for Eclipse on Linux for zSeries. This particular Eclipse installation uses the M5 Linux-Motif stable build from April 16, 2002. This build and all other builds for Eclipse are found under the download section at:

<http://www.eclipse.org/>

9.1 Prerequisite software for Eclipse

Eclipse requires a Java Runtime Environment (JRE) or Java Development Kit (JDK) at version 1.3 or higher. The JDK installation instructions are discussed in 2.2.2, “Installing the IBM Java Developer Kit” on page 28.

Given that our Eclipse download is for Motif, we need Motif or a Motif substitute plus QT in order to display Eclipse’s Graphical User Interface (GUI) from the Linux host to the client machine. Our Motif substitute is Openmotif. Openmotif and QT are standard packages that come with the SuSE distribution and they are installed as part of the standard installation.

In addition, we need a Xserver installed on the client machine to complete the GUI display connection. Examples of common Xservers are Hummingbird Exceed and Cygwin/XFree86, which is a port of XFree86 to Cygwin. The Hummingbird Xserver can be found at:

<http://www.hummingbird.com/>

The XFree86 server for Cygwin can be found at:

<http://xfree86.cygwin.com/>

The following packages were used in our setup to start our installation of Eclipse:

- ▶ JDK version 1.3
- ▶ Openmotif version 2.1.3
- ▶ QT version 2.3.0

9.2 Eclipse installation

The binaries that come with Eclipse do not work for Linux on zSeries at this time. To complete the installation, the C source distribution must be recompiled. The following sections describe the steps needed to compile and install Eclipse and the Standard Widget Toolkit.

9.2.1 Rebuilding Eclipse

Note: These commands are specific to the M5 Linux-Motif stable build and assume the `ecclipse` executable will be written to the `/usr/local/bin/ecclipse` directory. We suggest that you place Eclipse in this directory for common use by multiple users. However, the installation instructions can be used for a stand-alone installation as well. Make adjustments accordingly if there is a different build or if Eclipse is unpacked to a different directory

To rebuild from source, perform the following steps:

1. Unzip the source package:

```
unzip ecclipse-SDK-20020416-linux-motif.zip -d /usr/local
```

2. Add `/usr/local/ecclipse` directory to the `PATH` environment variable:

```
export PATH=/usr/local/ecclipse:$PATH
```

Note: We suggest that you update the `/etc/profile.local` file with this command to make the command permanently available to users.

3. Unzip the launcher source to the `/usr/local/ecclipse` directory:

```
unzip launchersrc.zip
```

4. Compile the launcher source:

- a. Change directory to the extracted `library/motif` directory:

```
cd library/motif
```

- b. Customize the `make_linux.mak` file, assigning the `MOTIF_HOME` variable the appropriate value.

Note: For our installation, this was `MOTIF_HOME=$(X11_HOME)`. This variable should point to the directory containing the `libXm.so.2` file.

- c. Build the source tree. From `/usr/local/ecclipse`, issue:

```
./build.csh
```

Note: Be sure the script is executable before issuing this command.

- d. Replace the original `ecclipse` with the newly created binary:

```
cp ecclipse ../..
```

9.2.2 Build the Standard Widget Toolkit

The Standard Widget Toolkit (SWT) needs to be compiled to be compatible with the newly installed Eclipse binary. The following steps outline the process:

1. Unzip the SWT source files in the SWT library directory:

```
cd /usr/local/eclipse/plugins/org.eclipse.swt/ws/motif
unzip swtsrc.zip
```

2. Customize the `make_linux.mak` file. Table 9-1 summarizes the required variables and the values we used.

3. Build the library:

```
./build.csh
```

Note: Be sure the script is executable before issuing this command

Attention!: If Gnome or KDE is not installed, this command will display error messages. These can be ignored provided the `libswt-motif-2034.so` module has been successfully created.

Table 9-1 Customized `make_linux.mak` variables for SWT build

Variable	Points to	Value used
IVE_HOME	Java home directory	/opt/IBMJava2-s390-13
MOTIF_HOME	Motif home directory	/usr/X11R6
QT_HOME	Qt home directory	/usr/lib/qt-2.3.0

9.3 Set up the environment

1. Make the `swt-motif-2034.so` module available to the `$LD_LIBRARY_PATH`. This can be done by either:

```
ECLIPSE_ROOT=/usr/local/eclipse
ECLIPSE_MOTIF=$ECLIPSE_ROOT/plugins/org.eclipse.swt/ws/motif/library
export LD_LIBRARY_PATH=$ECLIPSE_MOTIF:$LD_LIBRARY_PATH
```

or

Copy the `swt-motif-2034.so` module to a path where it can be found (for example, the JDK's `jre/bin` directory).

Note: We suggest that you update the `/etc/profile.local` file to make the change available to all users.

2. Export the `DISPLAY` variable to the Xserver IP address from the client on the Linux host with the following command:

```
export DISPLAY=Xserver_IP_Address:0.0
```

Note: The `Xserver_IP_Address` is the IP address of the client machine. Therefore, this command should be done independent for each user. We suggest the users update their own `.profile` with this command to make the setting permanent. Make sure the client Xserver is running.

3. Remove the Motif library references that come with Eclipse under the `/usr/local/eclipse` directory with the following commands:

```
rm /usr/local/eclipse/libXm.so
rm /usr/local/eclipse/libXm.so.2
rm /usr/local/eclipse/libXm.so.2.1
```

9.3.1 Testing the installation

Make sure the Xserver is running on the client machine.

From the Linux host, type the following command at the Eclipse installation root directory:

```
eclipse -vm Java_Jre_Bin_Command
```

where `Java_Jre_Bin_Command` is the Java command located on the JDK's `jre/bin` directory.

In our installation this command looks like:

```
eclipse -vm /opt/IBMJava2-s390-13/jre/bin/java
```

At this time the Eclipse initial GUI screen should appear on the client machine.

9.4 Installing the C/C++ Development Tools plug-in

The Eclipse Software Developer Kit (SDK) comes with the platform, the Java Development Toolkit (JDT) plug-in, and the Plug-in Development Environment (PDE). The JDT provides a full development environment for Java. The PDE provides a way to develop Eclipse plug-ins while inside the Eclipse workbench.

In order to achieve a C/C++ development environment, the C/C++ Development Tools (CDT) Plug-in needs to be installed. The code for this plug-in can be downloaded from the projects option at:

<http://www.eclipse.org/tools/index.html>

The CDT consists of two parts: server and client. What we mean by that statement is that code can be remotely developed from a client machine. If code is developed remotely from a client machine, the server part of the CDT needs to be installed on the host and the client code installed on the client. On the other hand, if code is developed locally, then only the client part of the CDT needs to be installed. Figure 9-1 shows one configuration that is possible using this setup.

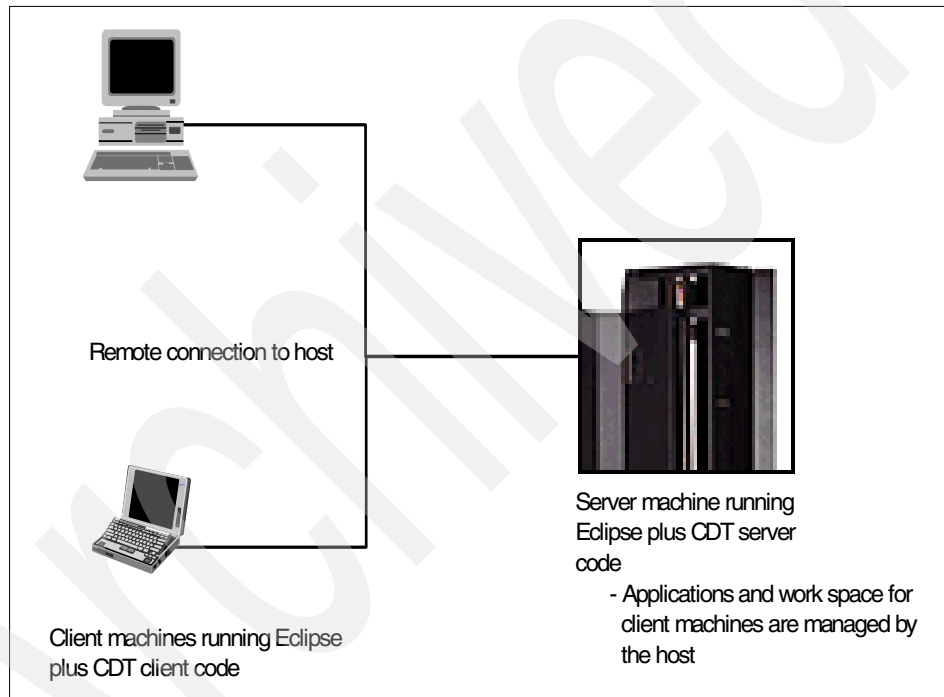


Figure 9-1 CDT client/server configuration

Note: These installation instructions refer to the CDT download from April 8th, 2002. Adjust the commands according to the download you use.

To start the installation after downloading the code, type the following command:

```
unzip cdt-eclipse-R2-20020408.zip
```

Note: This command makes and unpacks the installation files under the cdt directory under the current directory.

9.4.1 Installing the CDT client

Follow these instruction to install the CDT client code:

1. Go to the directory where the CDT is unpacked. If the current directory is where the cdt-eclipse-R2-20020408.zip file exists, type:

```
cd cdt
```

2. Unpack the client code to the Eclipse plugins directory. Under our setup this command is:

```
unzip cdt-eclipse-R2-20020408-local.zip -d /usr/local/eclipse/plugins
```

At this time the Eclipse workbench should be able to create C/C++ projects from the local machine.

9.4.2 Installing the CDT server code

If some people need to work remotely from the host, the CDT server should be installed. The next set of instructions shows how to accomplish this task.

1. Go to the directory where the CDT is unpacked. If the current directory is where the cdt-eclipse-R2-20020408.zip file exists, type:

```
cd cdt
```

2. Unpack the server code to the plugins directory under the installation directory for Eclipse. Under our setup this command is:

```
unzip cdt-eclipse-R2-20020408-server.zip -d /usr/local/eclipse/plugins
```

Note: Respond **A** to replace all files if a prompt to replace any file is displayed. This can occur if the client code is installed and later it is decided that the server code is needed.

3. Start the DSTORE server by running the server.linux script. Make sure the PATH variable contains the IBMJava2 bin directory. The CLASSPATH variable needs to be updated to include the server files. In our setup these variables are set and the script is started with the following commands:

```
export PATH=/opt/IBMJava2-s390-13/bin:$PATH
PLUGINS=/usr/local/eclipse/plugins
DSTORE=org.eclipse.cdt.dstore
CPP=cpp.miners
J1=$PLUGINS/$DSTORE.extra.server/extra_server.jar
```

```
J2=$PLUGINS/$DSTORE.core/dstore_core.jar
J3=$PLUGINS/$DSTORE.miners/dstore_miners.jar
J4=$PLUGINS/$CPP.miners/cpp_miners.jar
J5=$PLUGINS/$CPP.miners.parser/miners_parser.jar
export CLASSPATH=$J1:$J2:$J3:$J4:$J5:$CLASSPATH
$PLUGINS/$DSTORE.core/server.linux &
```

Once the DSTORE server is started, the DSTORE daemon needs to be started with the `daemon.linux` script to allow for user connections (run as root to allow for user authentication). In our setup, it could be started using:

```
/usr/local/eclipse/plugins/org.eclipse.cdt.dstore.core/daemon.linux &
```

Note: We took the default port (4033) to accept the connections. If the port number needs to be changed or the procedure fails, look at the `server.htm` file for further information. In our setup this file is located in directory:

```
/usr/local/eclipse/plugins/org.eclipse.cdt.cpp.docs.user/tasks/remote
```

Configuring Eclipse

This chapter describes some guidelines, ideas, and facts to keep in mind while working with Eclipse. They deal specifically with the way Eclipse works and behaves under our particular stable build. Future builds and releases of Eclipse may behave differently.

10.1 Starting Eclipse

There are some initialization parameters for starting Eclipse that are necessary to launch the application. Some others are necessary for Eclipse to behave properly under some circumstances. In this section we discuss some of these parameters, plus a small procedure to make life easier.

10.1.1 The `-vm` option

In 9.3.1, “Testing the installation” we use the `-vm` parameter to start Eclipse. This parameter is needed in order to specify the Java VM to run the Eclipse platform. The location defaults to the `../jre/bin/java` directory (relative to the executable) if the command line option is not specified. This is not the case in our setup, so we have to type the option as a command line argument. This option is also important if a different Java VM needs to be specified either for support or development reasons.

10.1.2 The `-data` option

The `-data` option specifies the working data directory for the running instance of Eclipse. This option becomes important in a multi-user environment where there is a common executable. By allowing developers to specify this option, we can ensure that a developer does not inadvertently corrupt or use the wrong project. By default, Eclipse creates and uses projects under the `.metadata/.plugins/org.eclipse.core.resources/.projects` directory under the current working directory if the `-data` option is not specified.

10.1.3 The `-vmargs`

The `-vmargs` option allows the user to pass a parameter directly to the Java VM. For some Java VMs, passing a parameter with this option may be critical for Eclipse to behave correctly. An example of a Java VM in this situation is the IBM Developer Kit, Java™ Technology Edition VM. It is recommended to call eclipse with the `-vmargs -Xmx256M` arguments to allow the Java heap to increase to 256 MB. This heap is usually recommended for large projects.

10.1.4 Other start options

The previously described options are the ones we consider to be of interest. There are other options that can be specified to modify the behavior of Eclipse. A complete list can be found in the Running Eclipse subsection under the Working with a Team section of the Workbench User Guide online help.

10.1.5 Simplifying options

A simple trick to get away from constantly typing startup options is to build a simple script file. Following is a simple procedure to accomplish this task.

1. Make a file called “myeclipse” with an editor like vi with the following command:

```
vi myeclipse
```

2. Type eclipse and follow it with the parameters. For simplicity, let's follow some of the installation parameters in our installation section. Then save and close the file.

```
eclipse -vm /opt/IBMJava2-s390-13/jre/bin/java -data $HOME -vmargs /  
-Xmx256MB
```

3. Attach the directory where the myeclipse file resides to the PATH environment variable. If the file is under the current directory this can be done with the following command:

```
export PATH=$PWD:$PATH
```

4. Make the file executable with a command such as:

```
chmod 755 myeclipse
```

Let's analyze what we are doing. We can call the myeclipse file from anywhere by making it executable and by attaching its path to the PATH environment variable. When the myeclipse script is called we are calling Eclipse with our own parameters. In this case, we tell it where to find the Java VM. We also indicate we want to create or use the projects under the \$HOME/.metadata/.plugins/org.eclipse.core.resources/.projects directory, and that we want to allow the Java VM to increase its heap to 256MB.

10.2 Configuring Eclipse to use CVS

Eclipse works with CVS to provide a team development environment. In this section, we describe a procedure for achieving a working environment between Eclipse and CVS.

Eclipse and CVS, in a team environment, work on a principle called repositories. A *repository* is a location where a code structure is stored. The team members can then access the code in the repository in order to work with it. Eclipse uses source code control mechanisms that are known to CVS to prevent conflicts that arise when multiple people work on a project, as well as code release mechanisms.

The first step to work with a CVS repository with Eclipse is to tell Eclipse where the repository is located. We found some discrepancies on how to achieve this in the current documentation, so this procedure may come in handy.

1. Bring up the CVS Repository Exploring Perspective. This is done through the Window menu option. Click **Window -> Open Perspective -> CVS Repository Exploring**.
2. Right-click in the working space of the CVS Repositories view. Select **New -> CVS Repository Location**.
3. This brings up a screen similar to Figure 10-1 on page 127. The screen shot contains sample parameters particular to our installation for CVS. For more information on CVS and our setup, look at 3.3, “Administering CVS” .

Following is a quick description on the parameters:

- Connection type can be pserver, ext, or extssh. In our CVS host setup we use ssh as our authentication method. In this scenario we can either use ext or extssh as our connection type from Eclipse. An ext connection from Eclipse prompts the user for a userid and password every time a connection is made to CVS from the place that Eclipse is launched. A way to prevent the prompts for userid and passwords is to specify an extssh connection from Eclipse. However, be aware that this connection is only known by Eclipse and not CVS. The CVS administrator can provide the information on the authentication method used by CVS.

Note: Extssh is only known to Eclipse. This authentication method is not known to CVS. CVS commands are not recognized by CVS when issued on the command line when this mode has been established by Eclipse. For example, if an extssh connection has been established by Eclipse, a project has been checked out and resources has been changed in Eclipse. See 11.4, “Using Eclipse with CVS” . The following CVS command will not work:

```
cvs diff
```

This command produces a message similar to:

```
cvs diff: unknown method in CVSroot:  
:extssh:eliuth@cvshome:/var/cvs  
cvs [diff aborted]: Bad CVSROOT.
```

- User and Password are the user ID and password of a user that has been given authority to access the CVS repository. This information can be obtained from the CVS administrator.

- Host is the name of the host machine that contains the CVS repository. The user can specify the IP address of the CVS host machine as well, and this may be the only way when problems with the DNS server arise.
 - The connection port is the port through which the CVS server listens for connections. If the default port does not work, contact the CVS administrator to find the connection port. Chances are that the connection port is not correct if the user gets a message stating that “Eclipse cannot connect to host” while attempting the connection.
 - Repository path is the path of the root repository for CVS. Refer to 3.7, “Creating a project” to correlate the information that is displayed in Figure 10-1.
4. To test the connection, click the **Finish** button. At this time the CVS repository resources can be accessed from Eclipse.

The screenshot shows the 'Add CVS Repository' dialog box. The title bar says 'Add CVS Repository'. Below it, the subtitle is 'Add a new CVS Repository to the Repositories View'. The dialog contains the following fields and options:

- Connection type:** extssh
- User:** eliuth
- Password:** masked with asterisks
- Host:** cvshome
- Repository path:** /var/cvs
- Use Default Port:** checked
- Use Port:** (empty field)
- Validate Connection on Finish:** unchecked checkbox
- Buttons:** Finish, Cancel

Figure 10-1 Defining a CVS repository to Eclipse

10.3 Eclipse and editors

Eclipse uses default editors for code resources. The default editors are very helpful, especially for Java and C/C++ resources. However, there are many editors out there. Some editors are adored by code developers. Some others are equally despised. Eclipse provides a way to specify a favorite editor to edit resource files according to file types. Following is a procedure to have Emacs be the default editor for C files.

1. From the main menu select **Window -> Preferences**.
2. In the Preferences window select **Workbench -> File Editors**
3. In the File types pane select *.c to change the default editor for this type of files.
4. Click the **Add** button corresponding to the Associated editors pane.
5. In the Editor Selection window, select External Programs and click **Browse**.
6. Point the Filter file to where Emacs is located. In our installation, Emacs is located in /usr/bin; therefore, our Filter input is /usr/bin/*
7. On the files pane, select emacs and click **OK**.
8. Click **OK** under the Editor Selection window.
9. Select emacs from the Associated editors pane and click the **Default** button.
10. Click **OK** in the Preferences window to activate the change.

Open a file inside a project that has a file type of c to test the above procedure.

10.4 Modifying Eclipse

The appearance and behaviors of Eclipse can be modified to suit user taste and needs. The procedure described in the previous section is an example of such a modification.

10.4.1 Workbench

Changing the default editor for a particular file type is an example of a workbench configuration change. The workbench has other selections that allow for changes as well. It is not our intent to go over every single selection since they are self explanatory, documented in the online help, and follow the same procedure described in 10.3, "Eclipse and editors" .

10.4.2 Perspectives and components

The behavior and appearance of perspectives and components can be changed the same way that workbench can be configured.

The following items can be configured for specific behavior:

- ▶ Ant
- ▶ C/C++
- ▶ Debug
- ▶ Help
- ▶ Java
- ▶ LPEX Editor
- ▶ Plug-In Development
- ▶ Team

Each of these items have a list of selectable sub-items. Each can be configured using the same procedure described in 10.3, “Eclipse and editors” .

Eclipse as an integrated development environment

In this chapter we discuss Eclipse as a integrated development environment (IDE). We start by looking at Eclipse concepts through the workbench. Then, we advance our understanding by exploring the JDT, PDE, and CDT as self-contained IDEs for Java, Plug-in, and C/C++ development. Along the way, we demonstrate the use of other tools that can be integrated to enhance the functionality of Eclipse.

11.1 Concepts

This section focuses on concepts that are essential to understand the Eclipse working environment. We keep the discussion simple to provide a quick overview. The online documentation provides more detailed information for those who want to investigate further.

11.1.1 Workbench

The workbench is the window that is visible when Eclipse is running.

11.1.2 Perspective

Perspectives exist inside the workbench. The perspective is the working environment within the workbench. Examples are the resource and JDT perspectives.

11.1.3 View

Views exist inside a perspective. We can consider a view as a pane inside a perspective. An example of a view is the navigator area which displays information about the structure of projects.

11.1.4 Editors

Editors are usually found inside the editor view. They help the user manipulate, edit, and change resources.

11.1.5 External editors

External editors are found outside the workbench. Eclipse provides a way to specify a favorite editor for resource manipulation. More information can be found in 10.3, “Eclipse and editors” on page 128.

11.1.6 Resources

Resources in Eclipse can be mapped directly to a directory file structure. Eclipse resources are known as projects, folders, and resources. In a directory structure, a project is similar to a directory, a folder is similar to directory inside a directory, and a resource is similar to a file.

We can view this structure in another way. A project is the parent, which can contain folders. A folder is a child of a project; it can contain other folders or resources. A resource is a child of a folder.

11.1.7 Graphical concept view

Figure 11-1 illustrates the concepts in this section.

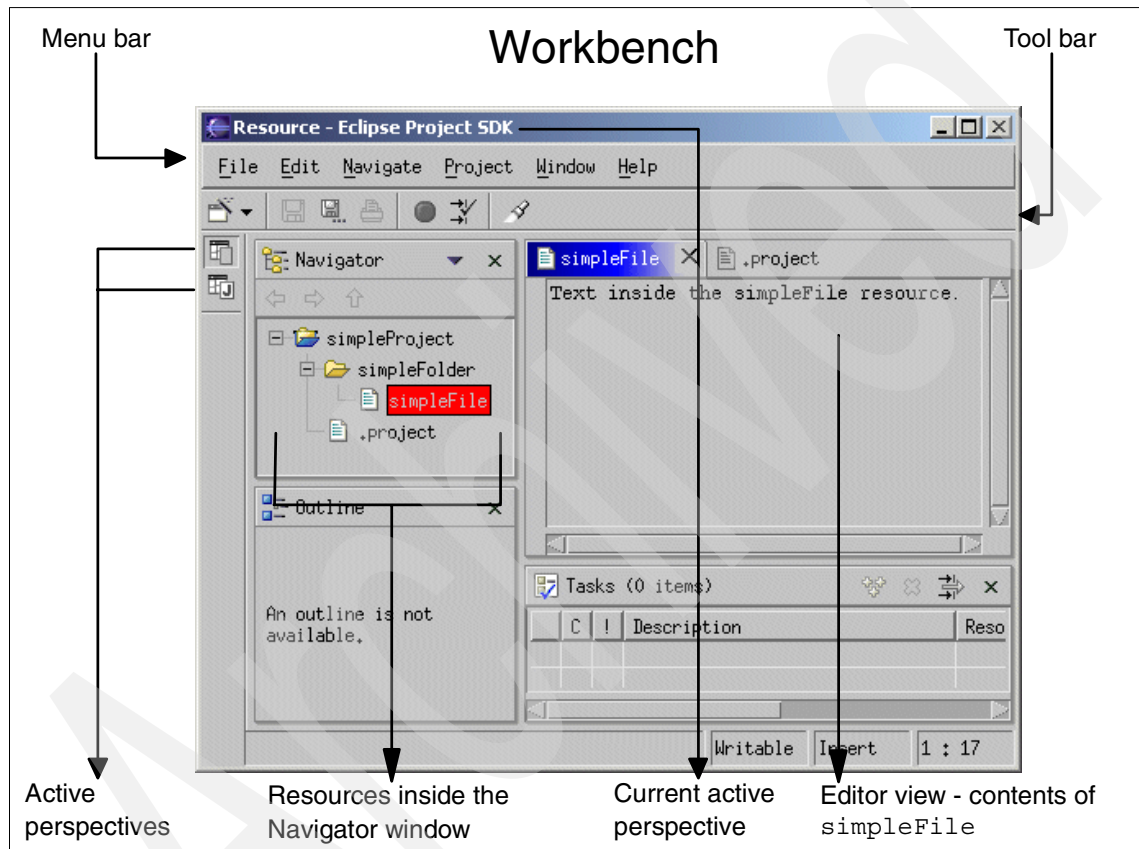


Figure 11-1 Graphical concept view

With respect to Figure 11-1, the workbench is the displayed window. The current perspective is the Resource perspective. We know that because the window title tells us. There are two active perspectives on the workbench: the Resource and Java perspectives. We know that because of the icons on the left side. There are four active views inside the Resource perspective: the Navigator, Outline, Editor, and Tasks views. We can see the resources simpleProject, simpleFolder, simpleFile, and .project file inside the Navigator view. We see that two

resources are opened by the editor: `simpleFile` and `.project`. We can further see that the current resource being edited is `simpleFile` because its folder tag inside the editor view is active.

11.2 Using the Java Development Toolkit

In the previous section we introduced the principles upon which Eclipse works. We can see the structure of a project at the resource level. However, notice that a resource is just a resource to Eclipse. Resources do not produce any special treatment at this level.

Now we turn our attention at the idea of a plug-in. A plug-in is an extension to the platform that enhances its functionality. The JDT is a plug-in that comes as part of the Eclipse installation and that enhances the functionality of the platform. In this case it offers an IDE for the development of Java applications. So what is an IDE, one may ask? An IDE is simply a development environment that allows developers to design, create, build, debug, and distribute applications within a working environment. Some of these functions may be missing in some IDEs, but that is the basic idea. Armed with that definition, we now look at the JDT as an IDE.

11.2.1 Menu bar and tool bar

In Figure 11-1 on page 133 we can see the menu bar for Eclipse. Start the workbench if it is not already started. Activate the Java perspective by making the following selections from the menu bar:

Window -> Open Perspective -> Java

A menu bar similar to the one on Figure 11-2 should be displayed.

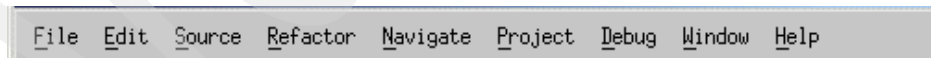


Figure 11-2 JDT menu bar

When you compare the two figures, you see that the Java perspective has added new functionality to the workbench. There are three new options on the menu bar:

- ▶ Source allows the Java developer to add code functionality and it allows a fast way to comment code.
- ▶ Refactor provides functions to quickly make system-wide code changes without changing its initial behavior.
- ▶ Debug provides a test environment to check code.

From our initial definition of an IDE we can see how this is starting to make sense. The JDT is a plug-in since it adds functionality. The definition can be extended to assert that an IDE for Eclipse is a plug-in, but not all plug-ins are IDEs.

We see similar changes when we compare the tool bars. This makes sense since tool bars are usually shortcuts to functionality made available inside a menu bar option.

11.2.2 JDT initialization

When working on a project, make certain that your environment is set properly. Otherwise, the code may not work properly or it may not compile. There are two variables in the preference window that affect the behavior of the JDT. These are the classpath and the installed JREs variables.

The classpath variables instructs Eclipse to look for JAR files in the path specified by these variables. Eclipse has three default classpath variables: JRE_LIB, JRE_SRC, and JRE_SRCROOT. The path inside these variables is decided by the path of the JRE being used. The user can add a variable to the classpath and thus extend it. If code does not compile because it cannot find a function or resource, this may be a good place to look for the problem.

Installed JREs variables can alter the way Eclipse behaves. In the preference window the user can add a JRE location and select the JRE that Eclipse uses to build and run Java programs. This option come in handy when the user wants to test code under different environments or test functionality under different JREs. It can also be considered a debugging tool if one runs into problems.

To make changes to these variable use the following steps:

1. From the menu bar click **Window -> Preferences**.
2. Expand **Java** by clicking on the expansion sign (+).
3. Select the individual variables under **Java**.
4. When the variables are selected, a corresponding right pane is presented to make changes. Make the changes necessary using the functions in the right pane.

11.2.3 JDT Java project

Let's establish that this is not a tutorial on Java code development—there are plenty of books that deal with that subject. The online documentation that comes with Eclipse also talks about additional Java resources. We concentrate on functionality and usability of the IDE in this section.

Start by opening the Java perspective with the following selections:

Window -> Open Perspective -> Java

Create a project by using the following steps:

1. Click the **Create a Java Project** icon on the tool bar.
2. Enter the name Prj when prompted for a name and click **Finish**.
3. Click the **Create a Java Package** icon on the tool bar.
4. Enter the name pkg when prompted for a name and click **Finish**.
5. Select the pkg package in the Packages view.
6. Click the **Create a Java Class** icon.
7. Enter Class1 in the name field. Select Public in the modifiers section. Select public static void main(String[] args) and inherit abstract methods under the method stubs selections. Click **Finish**
8. Select pkg in the Packages view.
9. Click the **Create a Java Class** icon.
10. Enter Class2 in the name field. Select Public under modifiers and only select Inherit abstract methods in the Method stubs sections.
11. At this point the Packages view should look like Figure 11-3.

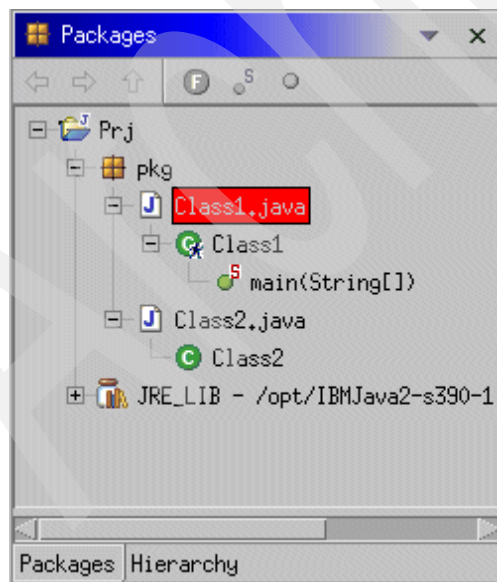


Figure 11-3 Packages view

We discuss this view later on this section. For now, we concentrate on our application.

We are going to build an application that shows the power of the IDE. We decided on a simple application that is composed of two classes. The first class shows some Java polymorphic characteristics with the myprint function. The correct function is called depending on the data type that is used. We use the other class to demonstrate the class structure between classes inside a package and projects. With that in mind, let's build a simple application.

1. Click the **Class2** icon in the Packages view. Notice that the IDE already generated a stub for the class plus header documentation.
2. Enter the code shown in Example 11-1 using the editor view. Note the code generation help capabilities that are available shown in Figure 11-4, which illustrates a case where the spelling of the println function is forgotten.

Example 11-1 Class2 code

```
public static void myprint(String parm) {  
    System.out.println("String parm = " + parm);  
}
```

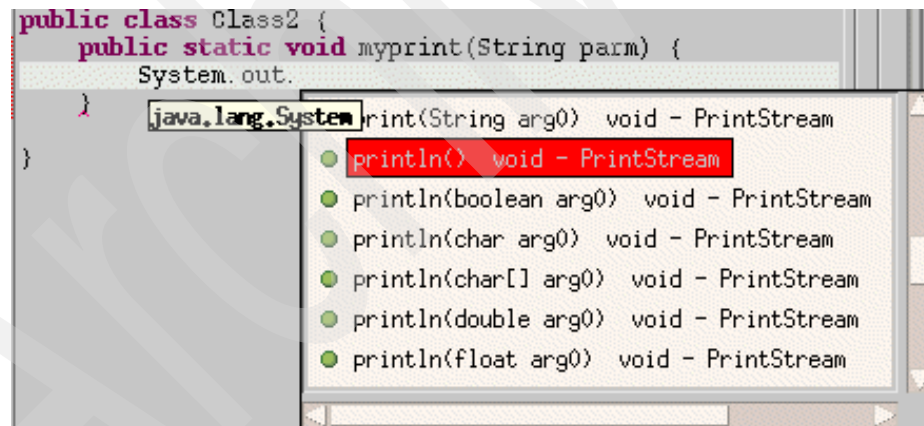


Figure 11-4 Editor help

3. Click the **Class1** class in the Packages view and enter the code shown in Example 11-2. The IDE detects there is something wrong, and signals the user that there is something wrong from the Packages view. Notice the x signs around the objects. The Editor view can provide further analysis of the problem: place the cursor on top of the offending function call to obtain this information. Another level of analysis is provided if you click on the x sign on the edge of the editor view next to the problematic function. When the x sign is

clicked on the Editor view, the Task window provides additional details about the problem. This is shown on Figure 11-5.

4. You can correct the problem by telling the code you explicitly want the method in Class2. Change the `myprint(stringData)` method call to `Class2.myprint(stringData)` in Class1.
5. Save the resource by right-clicking the Editor view and selecting the Save option. Note that this can be done as well by using the Ctrl-S shortcut key combination.

Example 11-2 Class1 code

```
public class Class1 {
    public static void main(String[] args) {

        int    intData    = 1;
        double doubleData = 1;
        String stringData = "Hello Redbook World";

        myprint(intData);
        myprint(doubleData);
        myprint(stringData);
    }
    public static void myprint(int parm) {
        System.out.println("Integer parm = " + parm);
    }
    public static void myprint(double parm) {
        System.out.println("Double parm = " + parm);
    }
}
```

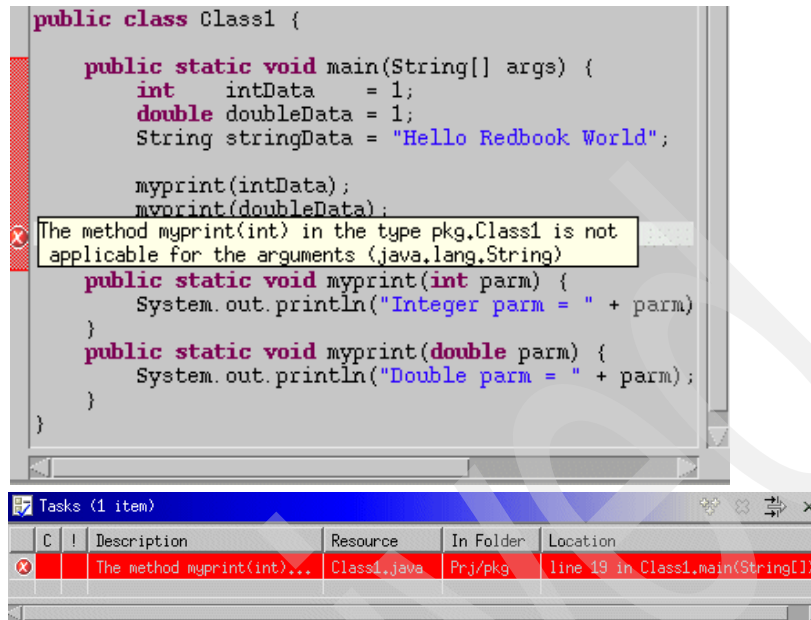


Figure 11-5 Editor task error view

11.2.4 Running the application

At this time the application should not have any errors. If it does, the following procedure will not work.

1. Make sure the Java perspective is active. The perspective is opened by clicking **Windows -> Open Perspective -> Java**.
2. Click the “running man” icon on the tool bar to start the launch configuration window.
3. Select Java application and click the **New** button.
4. Enter Redbook_configuration in the Name field.
5. Enter Prj in the Project field.
6. Enter pkg.Class1 in the Main class field.
7. Click the **Run** button.
8. The debug perspective is made active and the console view should look similar to Figure 11-6 on page 140.

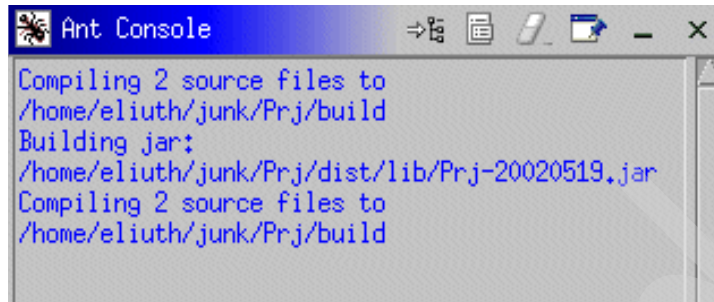


Figure 11-6 Console view

11.2.5 Debugging the application

Eclipse provides a debug environment for application development and testing, holding true to our definition of an IDE. The following procedure is geared to help the user debug the sample application.

1. Make the Java perspective active if it is not the current perspective by clicking **Window -> Open Perspective -> Java**.
2. In the Editor window, double-click the edge of the editor where a breakpoint is desired, as show on Figure 11-7 on page 141.
3. Select the Class1 object on the packages view. Click on the debug icon on the tool bar and select the Reedbook_configuration under the Java Application option on the launch configurations window. **Note:** The debug icon needs to be selected. If the run icon is selected the application only runs and the break point is not hit.

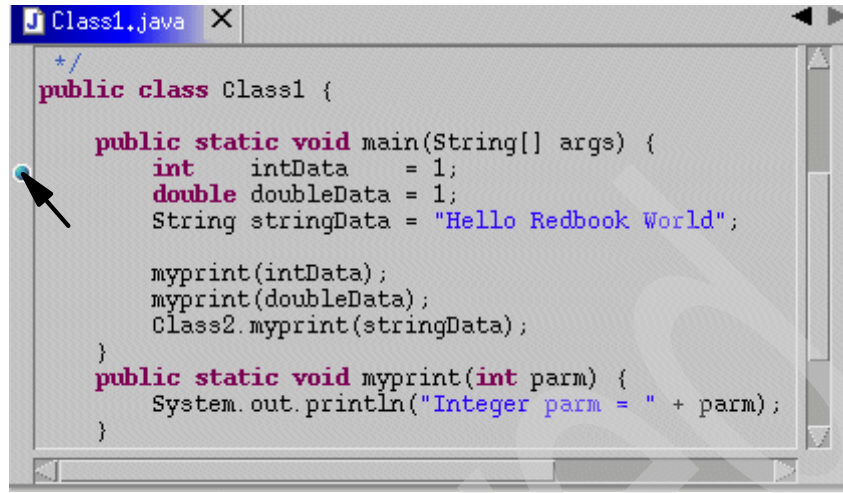


Figure 11-7 Setting breakpoints

4. Click the Debug button. Notice that the debug perspective is made current and the process is halted at the break point. Navigate the code using the debug toolbar provided on the debug view. Use the editor view to quickly check for variable values. See Figure 11-8 for examples. The other views in the perspective are designed to help debug the code by checking variables, setting break points, and checking expression within the code. For more information on debugging an application look at the Java Development User Guide online documentation for Eclipse.

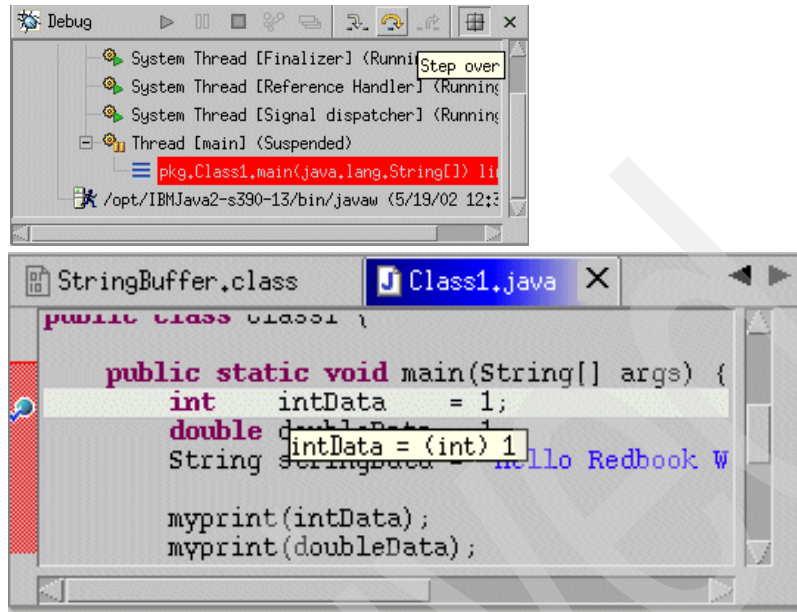


Figure 11-8 Debug and editor views

11.3 Using Eclipse with Ant

Eclipse, when generating plug-ins, automatically generates a build.properties file and a plug-in.xml file. The information contained in the build.properties file is used to create the plug-in.xml file, which in turn serves as input for Ant.

In our simple project example, these files are not generated. However, this does not stop you from using Ant with your sample project. The following procedure allows you to integrated Ant for compilation and distribution of your project:

1. Select the Prj project object on the Navigator view.
2. Right click and select **New -> file** to create a file under the Prj project.
3. Enter build.xml in the File name field on the New file window. Click **Finish** to create the file.
4. Select the build.xml file on the Navigator window and enter the code shown in Example 11-3 on page 143 on the Editor view for the file. The xml file provides four options for Ant when it is invoked. These options are clean, compile, dist, and init. Clean deletes the dist and build directories. Compile compiles the source code and places the compiled code under a build/pkg directory in relation to your Prj directory. Dist creates your distribution and

places it under the Prj/dist/lib directory. The file name that is created is Prj-timestamp.jar. The timestamp is created by init every time Ant is run.

Example 11-3 Ant script

```
<project name="Prj" default="dist" basedir=".">
  <description>
    Build Prj project
  </description>
  <!-- set global properties for this build -->
  <property name="sourceloc" location="pkg"/>
  <property name="buildloc" location="build"/>
  <property name="distloc" location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory -->
    <mkdir dir="${buildloc}"/>
  </target>

  <target name="compile" depends="init"
    description="compile source code" >
    <!-- Compile the java code from ${sourceloc} into ${buildloc} -->
    <javac srcdir="${sourceloc}" destdir="${buildloc}"/>
  </target>

  <target name="dist" depends="compile"
    description="generate distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${distloc}/lib"/>

    <!-- Put in ${buildloc} into the Prj-${DSTAMP}.jar file -->
    <jar jarfile="${distloc}/lib/Prj-${DSTAMP}.jar" basedir="${buildloc}"/>
  </target>

  <target name="clean"
    description="clean up" >
    <!-- Delete ${buildloc} and ${distloc} directories -->
    <delete dir="${buildloc}"/>
    <delete dir="${distloc}"/>
  </target>
</project>
```

5. Select the build.xml file and right-click it. Select **Run Ant** and select **Compile and dist** in the Run Ant window. Click **Finish**. The Ant Console view is displayed, with a result similar to Figure 11-9.

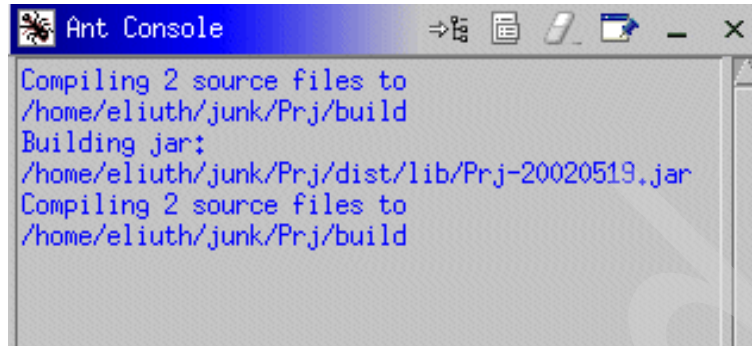


Figure 11-9 Ant console

This procedure is intended to quickly integrated Ant into Eclipse. Caution should be taken when working with plug-ins since the xml file for the build process is automatically generated. Manual build changes to the build.properties and plug-in.xml files can be undone by Eclipse when auto generating these files. There is a way to tell Eclipse to leave this file alone, but this is beyond the scope of this exercise. For further information, see the PDE Guide online documentation.

11.4 Using Eclipse with CVS

In a perfect world, we would all write code alone from our basements. Since this is not a perfect world, we have to work with others when writing applications. In a team environment, communication and understanding of the code structure is vital when working in a project.

Teamwork gives us the speed to accomplish major tasks in less time, but it comes at a price when handling code. One of these problems is that we need a common repository for all developers. Release and check-in mechanisms are vital to prevent people from overwriting one another's changes. Eclipse interfaces with CVS to minimize some of these problems.

The following procedure introduces CVS in combination with Eclipse as a source code management mechanism. Let's continue with the sample application to incorporate a team concept.

1. Make a connection to a CVS repository. See 10.2, "Configuring Eclipse to use CVS" on page 125 for instructions.
2. Make the Java perspective the current perspective and select the Prj project.
3. Right-click the Prj project and select **Team -> Share Project**.

4. Select the CVS repository in the Share Project window and click **Next**.
5. Select Use module name as project and click **Finish** to continue.
6. Select the Prj project item in the Synchronize - Outgoing mode view. Right-click it and select **Commit**. Enter a comment on the Commit Comment window and click **OK**.
7. Select the CVS perspective. Select the CVS repository. Right-click and select **Refresh View**. The Prj project should be displayed, as shown in Figure 11-10.

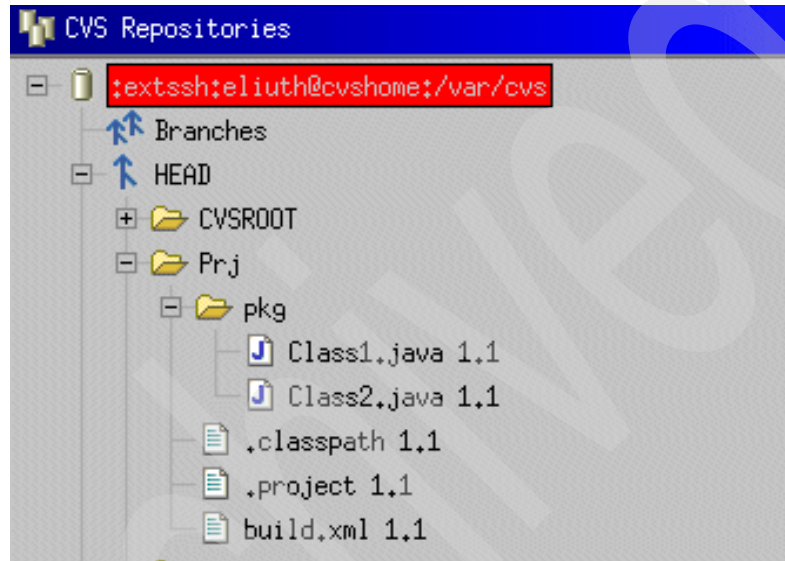


Figure 11-10 CVS Repository view

8. Bring up another instance of Eclipse and point it to another working directory with the **-data** command line option. The reason you want to do this is have a workspace that doesn't know about your Prj project. In essence, you are simulating another user. An example is:


```
eclipse -vm /opt/IBMJava2-s390-13/jre/bin/java -data $HOME/ws2
```
9. Open the CVS Repository Exploring perspective by clicking on **Window -> Open Perspective -> CVS Repository Exploring**.
10. Establish a CVS repository connection. Point this connection to the CVS that has the Prj project.
11. Expand the CVS repository and locate the Prj project. Check out the project, as shown on Figure 11-11 on page 146.

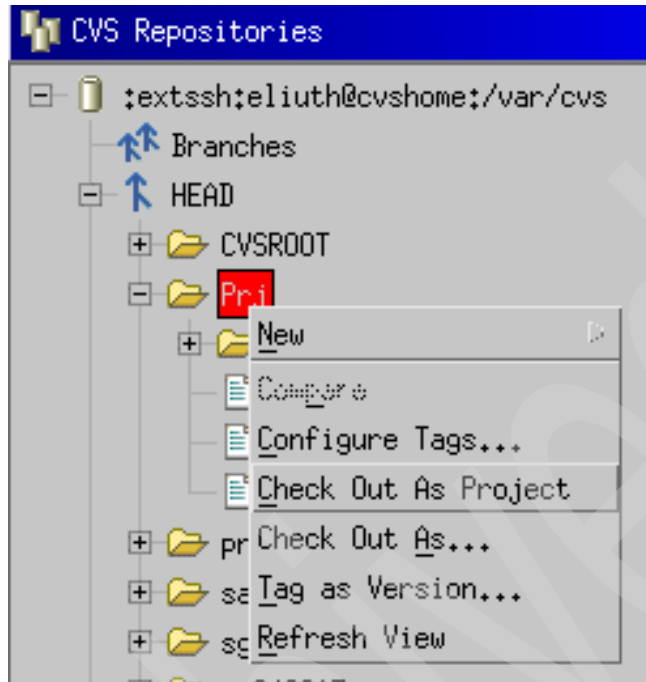


Figure 11-11 CVS check out

12. Make the Java perspective current by clicking **Window -> Open Perspective -> Java**.
13. Note that the Prj project is now on the Navigator view. Edit the Class1 class and enter a comment after the header documentation, such as the one displayed on Figure 11-12. Save the file after entering the change.

```
// Change made by another user
public class Class1 {
```

Figure 11-12 Comment change

14. Select the Prj project from the Navigator view and right-click it. Select **Team -> Synchronize with Repository**.
15. On the Synchronize - Outgoing Mode view, select Prj. Right-click it and select Commit. Notice that only the Class1 object is detected.
16. Enter a comment on the Commit Comment window and click **OK**.

17. Go to the previous instance of Eclipse. Note that the change is not on the Class1 resource.
18. Select the Prj project and right-click it. Select **Team -> Synchronize with Repository**.
19. Note that the Synchronizing - Incoming Mode view detects the change that was made to Class1.
20. Select the Prj project and right-click it. Select the **Update from Repository** option. After precessing the change, Class1 shows the change made by another user.

This is a small example on how to share resources. Other issues, like versioning control, are discussed in detail in the online documentation that comes with Eclipse. Refer to the online documentation for further information on how to use CVS to control team projects.

11.5 Using the C Development Toolkit

The C Development Toolkit (CDT) is an Eclipse plug-in that serves as an IDE for C and C++ application development. In this section we take a look at some sample code that comes as part of the CDT installation. In 11.2.3, “JDT Java project” on page 135 we demonstrated how to create and manipulate a project. The basic concepts are the same for the CDT except for a few name changes in the file structure. For example, in the JDT a directory is called a package. In the CDT a directory is called a directory. However, it is the same file structure.

11.5.1 Sample project

The CDT plug-in comes with a sample application called “payroll.” This application is discussed in detail in the online CDT documentation. The purpose of this exercise is to show the reader how a project is structured and the functionality that is provided by the IDE. Begin by bringing the sample payroll project into the IDE using the following procedure:

1. Bring up the CDT perspective by selecting **Window -> Open Perspective -> Other**, then choose C/C++ in the Select Probative window.
2. In the Navigator view, click the C/C++ Projects folder at the bottom of the view. The title of the view should change to C/C++ Projects.
3. Select **File -> New -> C/C++ Project** from the menu bar.
4. In the Create a new project resource window, enter payroll in the project name field.

5. In the Project resource location section there are three options:
- Default workbench location.
If this option is chosen the project is placed under a directory under the current working space. The name of the directory is the name of the project specified.
 - Local directory
This option allows the user to specify a directory location on the local machine where Eclipse is running.
 - Host
The Host option allows the user to work on a remote machine that is running the CDT server code. (Review “Installing the CDT server code” on page 121 if necessary.) The user can specify a remote machine if that machine is running the CDT server. The project resides on the host machine and the processing occurs remotely on the host. Figure 11-13 shows a host connection that is asking for authentication from the user to connect to the remote machine. Note that 4033 is the default port where the server code waits for connections. The login window is displayed when the browse button is clicked.

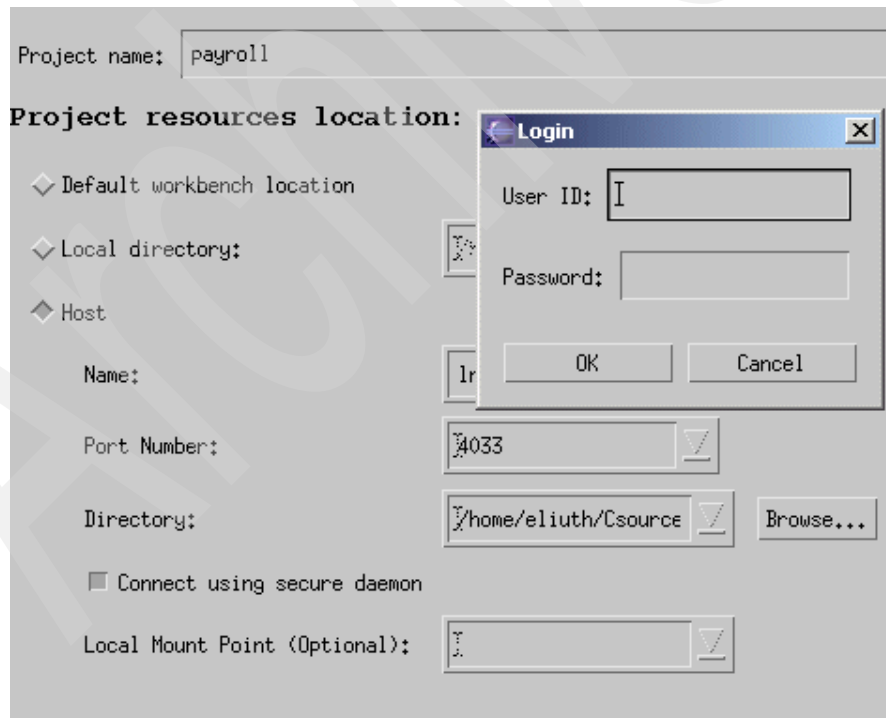


Figure 11-13 CDT host connection

Regardless of the connection being used, point the project location to where Eclipse is installed:

`./eclipse/plugins/org.eclipse.cdt.cpp.docs.user/sample/payroll.`

This is where the sample code resides.

6. Click **Finish**. The C/C++ view should look like Figure 11-14. From this window the user can look at the source code and make file.

Note: Lpex is the default editor for C and C++ file types. On our version of Eclipse on zSeries, Lpex core dumps Eclipse. Use the procedure described in 10.3, “Eclipse and editors” on page 128 to change the default editor for these files if you encounter this problem.

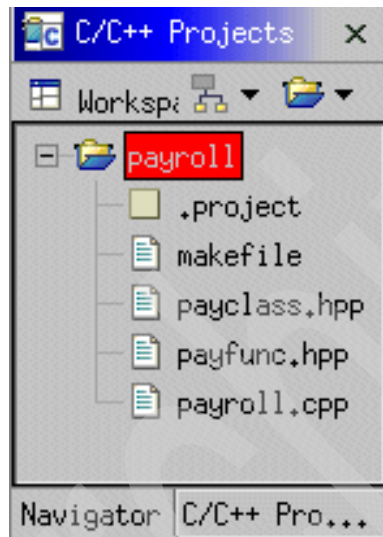


Figure 11-14 C/C++ payroll project view

11.5.2 Navigating code

You can double-click a specific file in the C/C++ Projects view and edit the file. A faster and easier way to navigate the code is by parsing it. Select the payroll project in the C/C++ Projects view and right-click it. Select **Parse -> Begin Parse**. After some processing the Project objects view displays the objects that are found in the code. From this view you can go directly to the code by double-clicking an object. Figure 11-15 illustrates this procedure. The items displayed in the Project object view can also be changed by clicking the double triangle icon in the view.

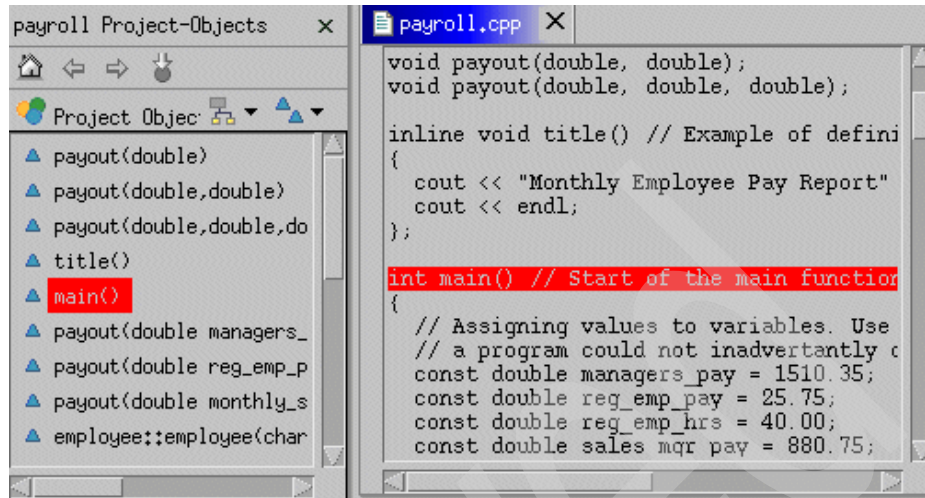


Figure 11-15 Object editor view

11.5.3 Compiling the project

To compile the project, select the payroll project in the C/C++ Projects view and right-click it. Select **Build project**. At this time the Output view shows the output of the build. If you have a host connection, this build is done at the host machine. Also note that after you build the project two new files appear on the C/C++ projects view. These files are payroll.o and payroll. These are the names specified inside the makefile when building the project.

11.5.4 Running the code

Use the following procedure to run the code:

1. Change the view to the Command Launcher view by clicking the Command Launcher folder icon on the command view.
2. Type payroll in the command input field and click **Run** to execute the application. Figure 11-16 shows this process.

Notice that the Output view contains the output from the application.

Tip: The Command Launcher can be used to input any valid Linux command; the output of the command is displayed on the Output window.

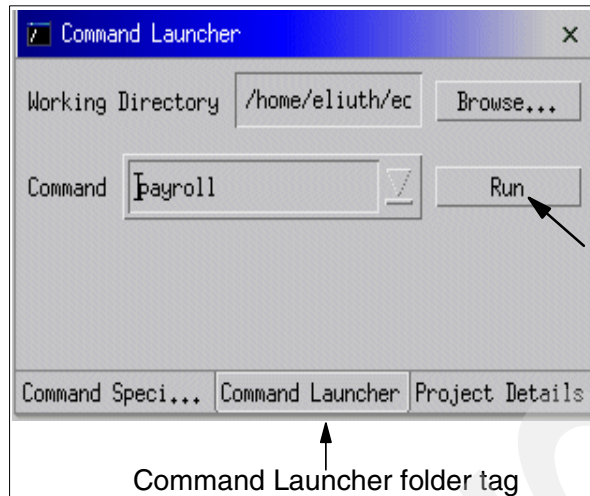


Figure 11-16 CDT running payroll

11.5.5 Debugging the application

We encountered some problems using the debugging capabilities documented by Eclipse with our version of the CDT. The main reason has to do with launching the Lpex editor in our version. However, this will not stop us from providing a work around. The following procedure describes how you can debug the payroll application.

1. Get a favorite debugger. In our case we used the Data Display Debugger (DDD), which can be downloaded from:
<http://www.gnu.org/software/ddd/>
2. Make sure the code is compiled with the `-g` compiler option in the makefile.
3. Go to the Command Launcher view and type `ddd payroll`. DDD should display the code inside its environment. Then DDD's functionality can be use to debug the code. Figure 11-17 shows DDD being used to debug the payroll application.

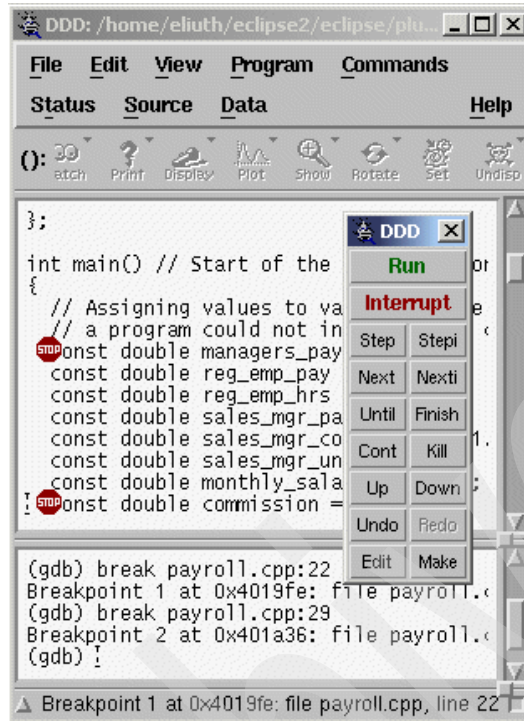


Figure 11-17 DDD view

11.5.6 Packaging and managing projects

The CDT uses autoconf to create and manage project configuration files. Using autoconf solves portability problems that occur on different system platforms. Be sure automake is found on the system before attempting to create configuration files.

To start autoconf for the payroll project, follow this procedure:

1. Select the payroll project from the C/C++ Project view and right-click it. Select autoconf and click **Configure**.
2. Choose Application for the type of deployment in the next window and click **Finish**.
3. At this time the configuration files for the project are built along with other files for documentation like AUTHORS, COPYRIGHT, Change.log, and INSTALL.

More information on autoconf and automake can be found at:

<http://www.gnu.org/manual/autoconf/>

For more information on managing projects see the CDT online documentation that comes with Eclipse.

11.6 Using the Plugin Development Environment

Eclipse provides a way to quickly develop plug-ins by providing the framework necessary. While plug-ins can be developed and deployed using the JDT perspective, this requires an extremely detailed understanding of the Eclipse framework. The Plugin Development Environment (PDE) provides an alternative, simpler way to create plug-ins. In this section we look at key components for developing a plug-in for Eclipse using the PDE perspective.

11.6.1 Setting up the development environment

Use the following procedure to set up the environment for plug-in development. If this is not done the generated code may not compile.

1. Select **Window -> Preferences**.
2. In the Preferences window, expand Plug-in Development and select Target platform.
3. In the Target platform window, choose the application radio button and click **Select All** to choose the required plug-ins for the application.
4. Click **OK** to accept the changes.

11.6.2 First plug-in

Use the following steps to build a plug-in using the templates already provided by Eclipse.

1. Activate the PDE perspective by selecting **Window -> Open Perspective -> Other**.
2. Select Plug-in Development from the Select Perspective window and click **OK**.
3. Select **File -> New -> Project**.
4. Select Plug-in Development and Plug-in Project in the New Project window and click **Next**.
5. In the next window, enter `myplugin` in the project name field and click **Next**.

6. In the next window, select Create a new plug-in using a code generation wizard, and select Hello, World for simplicity. Click **Next**. On the next window click **Finish** to accept the rest of the defaults.

At this time the Packages view should display the files that compose the project.

11.6.3 Making sense

To make sense of the procedure just completed, let's start by restating our definition of a plug-in. A plug-in is an extension to Eclipse that adds to its functionality. This functionality is usually made available through the workbench. Even though we are doing a simple Java `println` example called `Hello something`, we should expect to use the workbench in some shape or form. Eclipse provides a set of frameworks that allow developers to add to the workbench functionality. The `org.eclipse.ui.*` package contains many of the application program interfaces (APIs) to the workbench. In this example we implement some of these interfaces to make this happen. See the Platform Plug-in Developer Guide that is part of the Eclipse online documentation for an explanation of each API.

The Hello, World template that we used to create our application is a way to add an action to the menu bar of Eclipse. When the user selects the new menu, it provides the user with an action option. This option, in return, displays a window with a message when selected. Figure 11-18 shows a sample run of the application.

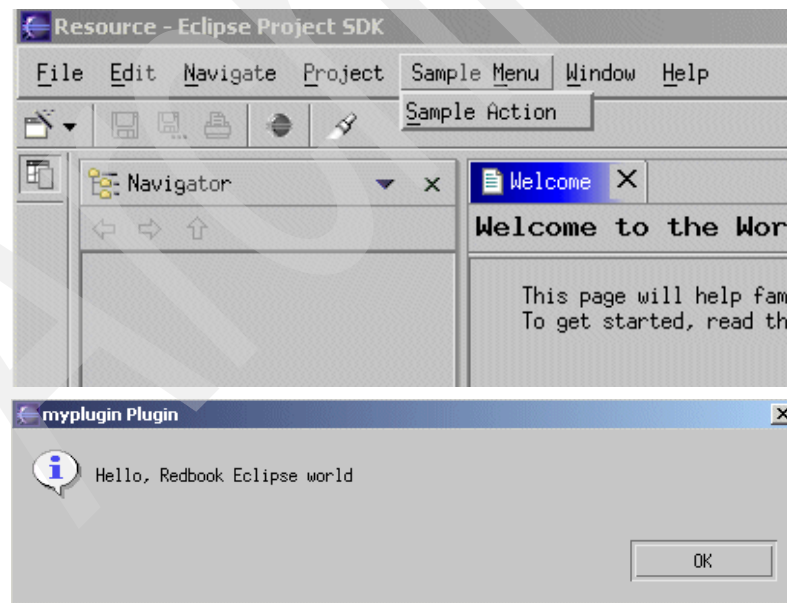


Figure 11-18 Sample run

Now we peek at how all this magic happens.

First we take a look at how the Sample Menu is created along with its Sample action. When we create a plug-in, a plugin.xml file is created. Extensions are specified using the graphical user interface for the plugin.xml file, but we are going to take a look at what is going on inside the file. View the source by double-clicking the plugin.xml file on the Packages view. Then in the Plug-in editor window click the source folder tag. You will see something similar to Figure 11-20.



Figure 11-19 Build.xml source

Notice that this is an extension to `org.eclipse.ui.actionSets`. If we look at the Platform Plug-in Developer Guide, we see that `actionSets` is the way that we add menus to the tool bar. Further analysis of the source shows that we have a menu and an action tag. Now all this is starting to make sense.

Now let's look at the action tag content. We see the label that is displayed when Sample Menu is clicked on the menu bar. We also see a class tag that references `myplugin.action.SampleAction`. This is the class that implements `SampleAction`.

If we look at the code for `SampleAction` and look at its `run` method, we should see the implementation. Figure 11-20 illustrates that is the case. It implements the action by displaying a dialog box with myplugin `Plugin` title and the message we want.

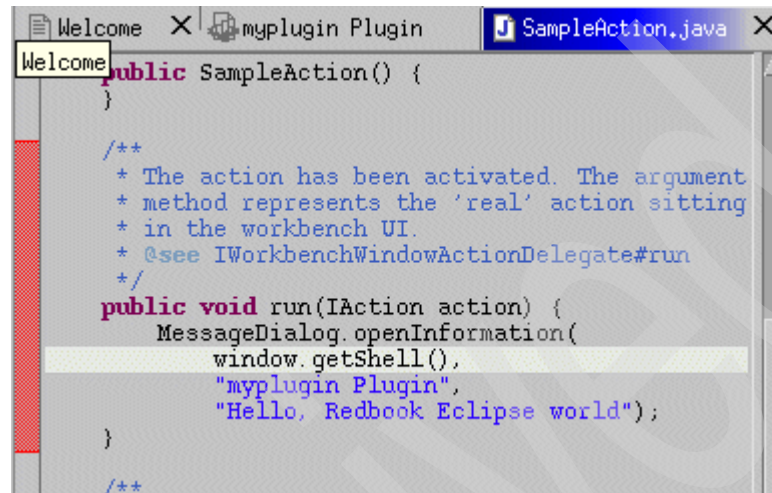


Figure 11-20 Plug-in implementation

11.6.4 Adding extensions

Now that we know how a plug-in works, we want to add extensions to one. Editing the plug-in file is not the recommended way to add extensions to a plug-in for several reasons. Not only is it hard work, but, more importantly, Eclipse may overwrite the changes when the project is built.

Start by viewing the `plugin.xml` file by double-clicking it. In the Editor view, select the extension folder tag. Figure 11-21 is a graphical representation of the code shown in Figure 11-19 on page 155 for extensions. At this time, you should have enough understanding of how to use the workbench to create a similar layout. If this is not the case, consult the PDE Guide online documentation that comes as part of the help for Eclipse. It has a detailed description of how to accomplish this task.

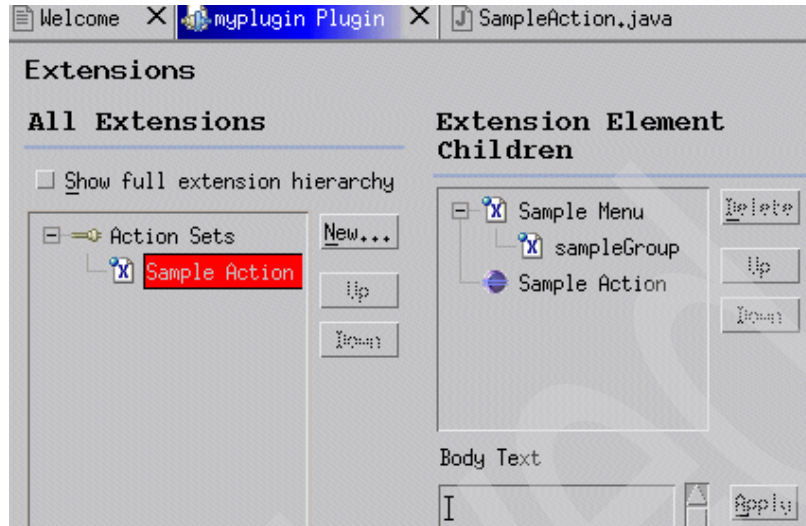


Figure 11-21 Plug-in extensions

11.6.5 Running the plug-in

The following procedure describes how to run the plug-in.

1. In the Packages view, select `myplugin` and right-click it.
2. In the Launch Configurations window, select Run-time Workbench and click **New**.
3. Click the Plug-ins and Fragments folder and select the Choose plug-ins and fragments to launch from the list option.
4. In the visible Plug-ins and fragments pane, click Select External Plug-ins and then click **Run**.

This procedure will launch another instance of Eclipse. The reason why we want to do that is to run a plug-in that is in the workbench but not yet installed. In essence, we want to quickly test the plug-in.

11.6.6 Deploying a plug-in

The PDE Guide that comes with Eclipse's online Help has a great section on how to deploy a plug-in. Refer to that section to learn how to deploy plug-ins for Eclipse distribution.



Part 3

Programming techniques

In this part of the book, we discuss programming techniques based on the tools presented in Part 1. We develop an address book sample application in two ways: as a Java Web application using Jakarta Struts, and as a C/C++ stand-alone application using the Qt library. Then we describe how to package and deploy the application.



zSeries as a development platform

In this chapter, we introduce an example application to demonstrate the Linux on zSeries development environment. We present an overview of the application, and describe the environment used to develop the application.

12.1 Example applications

To demonstrate some of the programming techniques presented here, we introduce two example applications:

- ▶ a Java Web application using Jakarta Struts,
- ▶ a C/C++ stand-alone application using the QT-Library.

These applications interact using the Java Native Interface (JNI). An architecture overview is presented in Figure 12-1.

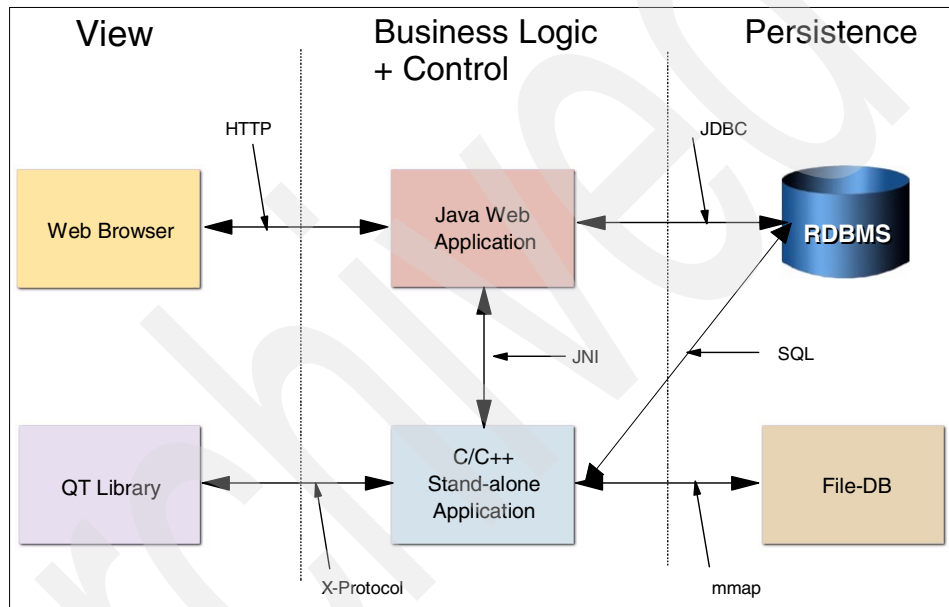


Figure 12-1 Application overview

12.1.1 Application overview

We examine several programming techniques and demonstrate those techniques by example:

- ▶ The Java Web application, discussed in detail in Chapter 13, “Using the Struts framework” on page 167, demonstrates application development using the Jakarta Struts framework. It utilizes two persistence mechanisms to store and retrieve data:
 - The JDBC interface.

- The Java Native Interface, used to invoke a C shared library. The JNI interface, in turn, utilizes static SQL to access the database (see Chapter 16, “Concurrency in embedded SQL” on page 237).
- ▶ The C/C++ application, described in detail in Chapter 14, “Shared libraries and more” on page 193, demonstrates several techniques specific to C programmers:
 - Using the Qt graphic libraries
 - Creating and using shared libraries
 - Using the `mmap` function

We discuss the application structure and how package the application for deployment in Chapter 17, “Packaging applications for deployment” on page 251.

12.1.2 The development environment

A major advantage of developing on Linux for zSeries is the opportunity it presents for consolidating the development environment onto a stable, centrally-managed platform. In a typical environment, application developers would not have root access on the machine. Installation and upgrades to development tools invariably requires root access. This can present logistical problems to the system administrator. For example, development machines may be widely dispersed, and tracking software levels across multiple machines becomes tedious.

In addition, application developers can face difficulties when working on a common development machine. For example, Web-based development requires a Web server. It is difficult for more than one developer to share a common Web server because, for example, tracing the cause of application errors is harder, and server restarts affect all developers.

Running Linux under zVM can help alleviate these problems because:

- ▶ Each developer can be assigned a Linux instance (minimizing adverse interaction between developers).
- ▶ System administrators can centrally manage all instances.

In Figure 12-2, we illustrate the development environment used in this book.

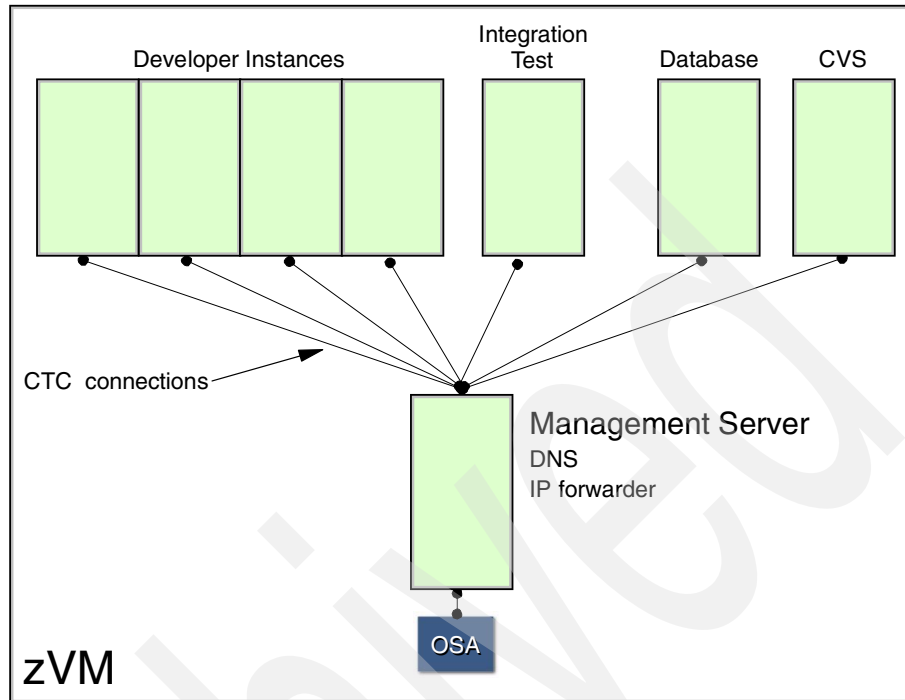


Figure 12-2 The application development environment

The development environment has the following characteristics:

- ▶ Each developer is assigned a Linux instance running under zVM. The full set of required development tools is installed on each developer's instance.
- ▶ One Linux instance is used for integration test.
- ▶ The database server is installed on a separate Linux instance.
- ▶ The CVS repository runs on a separate Linux instance.
- ▶ A management instance connects all Linux instances:
 - Linux instances exist in a distinct private subnet (10.1.1.0 for instance).
 - Virtual CTC interfaces connect Linux images to the management server.
 - The management server runs DNS for name and IP address resolution of the private subnet.
 - A single OSA interface (shared with zVM) connects the management server to the intranet. This conserves both hardware resources and intranet IP addresses (because the private subnet uses the reserved 10.1.1.0 network).

- Using the **iptables** command, the management server uses IP forwarding and network address translation (NAT) to route to and from the intranet. See Chapter 11, “Network infrastructure design” in *Linux on IBM @server zSeries and S/390: ISP/ASP Solutions*, SG24-6299 for a description of how to set up NAT.

Using the Struts framework

This chapter describes an example Web application created using the Jakarta Struts Web application framework. Topics included are:

- ▶ How to implement an application in the Struts framework
- ▶ Using Log4j to create logging messages
- ▶ How to use persistence in the Struts framework

13.1 The Struts application components

An address book Web application will be used to demonstrate the Struts application framework. The application has two address book functions: add a user and search for a user. Address book entries consist of users with a first name, a last name, an e-mail address, and a phone number.

The application consists of:

- ▶ Model components
 - Business Logic classes:
 - User to manage address book entries. Users are stored and retrieved by the persistence components, and acted on by the Struts Action classes.
 - Struts ActionForm classes:
 - AddUserFormBean to validate the addUser form.
 - SearchUserFormBean to validate the searchUser form.
- ▶ View components
 - JSP pages:
 - index.jsp - the application home page
 - addUser.jsp - a form to add users
 - searchUser.jsp - a form to search for users
 - error.jsp - a page to handle unexpected errors
 - log_error.jsp - an error logging page
- ▶ Controller components
 - Struts Action classes:
 - AddUserAction to process user additions
 - SearchUserAction to process user searches
- ▶ Persistence classes
 - ConnectionManager to manage database connections
 - AddressBook - an abstract class to hide the underlying persistence implementation
 - JdbcAddressBook - class to implement persistence using JDBC
 - JniAddressBook - class to implement persistence using the Java Native Interface (JNI) to a C shared library

For a graphical version of the Struts framework operation, refer to Figure 6-1 on page 85. We illustrate the framework using the *add user* scenario.

13.2 The model component

The model component is designed to isolate the business logic (as implemented in the User class) from the Struts framework. This decoupling allows an entirely different front-end (for instance, a Java application) to reuse the logic embodied in the User class.

Tip: Although not mandatory, isolating business logic from the Struts framework may promote reuse of your application classes. Whenever possible, consider well-defined interfaces between Struts and your business objects that decouple the framework from the application logic.

13.2.1 User class

The User class is implemented as a JavaBean and serves to mediate interaction between the persistence layer and the Struts framework. Example 13-1 illustrates its implementation.

Example 13-1 Application business logic (User.java)

```
package com.ibm.itso.sg246807.bo;

public class User implements java.io.Serializable {
    private String firstname = "";
    private String lastname = "";
    private String email = "";
    private String phone = "";
    public User(String first, String last, String email, String phone) {
        this.firstname = first;
        this.lastname = last;
        this.email = email;
        this.phone = phone;
    }
    public User() { }
    public String getLastname() { return lastname; }
    public void setLastname(String lastname) { this.lastname = lastname; }
    public String getFirstname() { return firstname; }
    public void setFirstname(String firstname) { this.firstname = firstname; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public String getPhone() { return phone; }
    public void setPhone(String phone) { this.phone = phone; }
```

```

    public String toString() {
        return new StringBuffer().append(firstname).append(", ")
            .append(lastname).append(", ")
            .append(email).append(", ")
            .append(phone).toString();
    }
}

```

Note: Struts components are not referenced or imported in this class; this isolates the User object from the Struts framework. The User class implements `java.io.Serializable` to allow it to be stored in an `HttpSession`.

13.2.2 ActionForm class

As introduced in Section 6.5.1, “Struts components” on page 84, applications derive from `ActionForm` to enable creation of JavaBeans from HTML form input in the framework. In Example 13-2, we show the implementation of an `ActionForm` used to validate submission of the `addUser.jsp` form.

Example 13-2 Struts ActionForm (addUserFormBean.java)

```

package com.ibm.itso.sg246807.form;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public final class AddUserFormBean extends ActionForm {
    private String firstname, lastname, email, phone;
    public ActionErrors validate(ActionMapping mapping,
        HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if (lastname==null || "".equals(lastname)) {
            errors.add("lastname",
                new ActionError("error.lastname.required"));
        }
        if (email==null || "".equals(email)) {
            errors.add("email",
                new ActionError("error.email.required"));
        } else {
            if (email.indexOf("@") < 0) {
                errors.add("email",
                    new ActionError("error.email.invalid", email));
            }
        }
    }
}

```

```

        return errors;
    }
    public String getFirstname() { return firstname; }
    public void setFirstname(String firstname) { this.firstname = firstname; }
    public String getLastname() { return lastname; }
    public void setLastname(String lastname) { this.lastname = lastname; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public String getPhone() { return phone; }
    public void setPhone(String phone) { this.phone = phone; }
}

```

The `validate` method ensures non-null, non-blank values for lastname and email have been provided in the form. In addition, it checks the email value for the presence of an '@' character (a rudimentary check for valid e-mail format).

The HTML form on which the `AddUserForm` class operates is shown in Example 13-4 on page 172.

13.2.3 Form validation and ActionErrors

The `validate` method performs validation prior to acting on the form (corresponding to step 4 illustrated in Figure 6-1 on page 85). Errors are returned as a collection of the `ActionError` instances. In the example, errors are referenced by a unique key pointing to the actual error message text. This is an important internationalization feature provide by Struts; further details about internationalization are in the next section.

13.2.4 Internationalization and application resources

Internationalization (often referred as *I18n*) is built in to the Struts framework. This mechanism is based on Java Internationalization features. For details on Java Internationalization, consult:

<http://java.sun.com/docs/books/tutorial/i18n/index.html>

To provide language independent messages, code messages in a resource properties file and refer to those messages by key. Example 13-3 shows the resource properties used by the `AddUserFormBean` class in Example 13-2 to display error message text.

Example 13-3 Resource properties (ApplicationResources.properties)

```

adduser.success=<li>User has been created.
adduser.failed=<li>User has NOT been created. Check log file for details.
searchuser.success=<li>User(s) retrieved.
searchuser.failed=<li>No users retrieved.

```

```
error.lastname.required=<li>Lastname is required.  
error.email.required=<li>Email is required.  
error.email.invalid=<li>Email "{0}" not valid.  
errors.header=Validation:<ul>  
errors.footer=</ul>
```

13.3 The view component

The view component presents HTML forms for user input and subsequent processing. The Struts framework provides custom taglibs to assist in the rendering of HTML. Here we examine the struts-html taglib in more detail.

13.3.1 Struts-html taglib

Using struts-html tags simplifies the process of creating ActionForm beans. Example 13-4 illustrates struts-html tags the framework will use in creating an addUserFormBean class in Example 13-2.

Example 13-4 Using struts-html tags (addUser.jsp)

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %> 1  
<html>  
<head>  
  <title>addUser.jsp</title>  
  <html:base/> 2  
</head>  
<body>  
  <h2>Add User</h2>  
  <html:errors/> 3  
  <html:form action="/adduser" focus="firstname"> 4  
    <table border="1">  
      <tbody>  
        <tr>  
          <td>Firstname</td>  
          <td><html:text property="firstname" size="20"/></td> 5  
        </tr>  
        <tr>  
          <td>Lastname</td>  
          <td><html:text property="lastname" size="20"/></td>  
        </tr>  
        <tr>  
          <td>email</td>  
          <td><html:text property="email" size="20"/></td>  
        </tr>  
        <tr>  
          <td>Phone</td>
```



```

        <td><html:text property="phone" size="20"/></td>
    </tr>
</tbody>
</table>
<br>
<input type="submit" name="adduserbutton" value="add">
</html:form>
<html:link href="/sg246807/">Home</html:link>
</body>
</html>

```

6
7

1. The `struts-html` taglib is defined to the JSP engine (see Section 6.4.2, “Configuring taglibs” on page 82).
2. The `html:base` tag renders to an HTML `<base>` element. This can simplify coding URLs on the page (by allowing all URL references to be relative).
3. The `html:errors` tag renders any errors reported in the `ActionErrors` collection returned by `AddUserFormBean` `validate` method. In the event no errors are returned, this tag renders to a null line.
4. The `html:form` tag renders an HTML `<form>` element.

Note: Struts will render the `action` attribute to the form:

```
action="/webapp/adduser.do;jsessionid=nnnnnnn"
```

This enables the `ActionServlet` to handle the request on form submission (see Section 13.6.3, “Configuring `ActionServlet`” on page 179).

5. The `html:text` tag renders an HTML `<input type='text'>` element.

Note: Input controls in the form (text, radio buttons, check boxes, and so forth) are created with `struts-html` tags specific to that element. Attributes on those tags map to their respective HTML options. The `name` attribute maps the element to the appropriate data member of the `ActionForm` bean. Be sure the `name` of these elements matches the corresponding data member in the `ActionForm` bean.

6. An HTML `</form>` element is rendered.
7. The `html:href` tag renders an HTML `<a href>` element.

13.3.2 Mapping form input to ActionForm beans

Figure 13-1 illustrates the rendering of struts-html tags to HTML elements and the mapping of those elements to the AddUserFormBean instance.

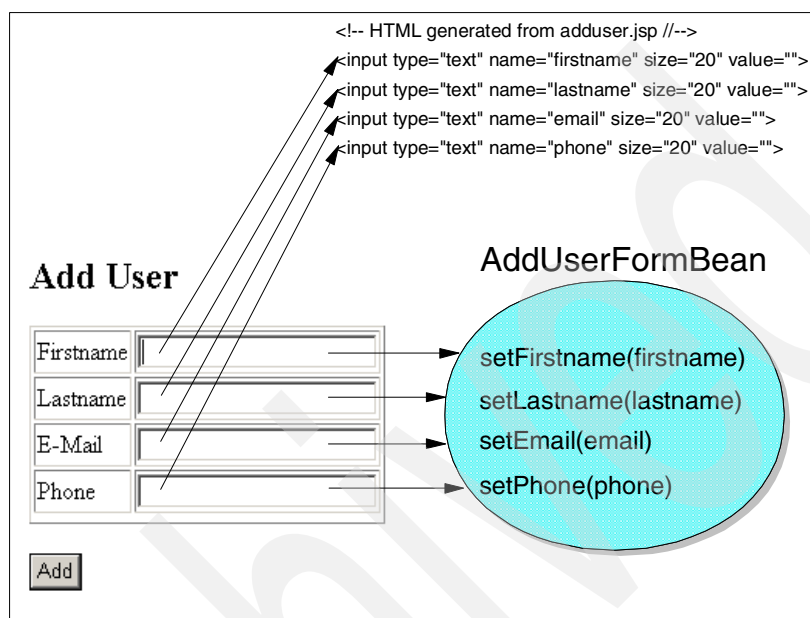


Figure 13-1 Mapping HTML form data to an ActionForm bean

13.4 The controller component

The controller component implements the application business logic. Much of the function is provided by the Struts framework. In this section, we illustrate how the *add user* function is developed, and how to configure the framework to recognize this function.

13.4.1 Action class

Handlers are developed by inheriting from the Struts Action class. When a form is to be processed, the appropriate Action instance perform method will be invoked by the `ActionServlet` instance. Example 13-5 shows the implementation that adds a user to the address book.

Example 13-5 Struts Action (addUserAction.java)

```
package com.ibm.itso.sg246807.action;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.ibm.itso.sg246807.data.Addressbook;
import com.ibm.itso.sg246807.bo.User;
import com.ibm.itso.sg246807.form.AddUserFormBean;
import org.apache.log4j.Logger;

public final class AddUserAction extends Action {
    Logger log = Logger.getLogger(this.getClass());

    public ActionForward perform(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response) {
        ActionErrors errors = new ActionErrors();
        User aUser = new User();
        AddUserFormBean bean = (AddUserFormBean) form;
        aUser.setFirstname(bean.getFirstname());
        aUser.setLastname(bean.getLastname());
        aUser.setEmail(bean.getEmail());
        aUser.setPhone(bean.getPhone());
        try {
            Addressbook book = Addressbook.getInstance();
            book.addUser(aUser);
            errors.add(ActionErrors.GLOBAL_ERROR,
                      new ActionError("adduser.success"));
        } catch (java.sql.SQLException ex) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                      new ActionError("adduser.failed"));
            log.error("searchuser.failed", ex);
        }
        saveErrors(request, errors);
        return (mapping.findForward("success"));
    }
}
```

Note: The `ActionForm` bean is passed as a parameter to the `perform` method. To access the specific bean for the handler, it is necessary to typecast to the appropriate `ActionForm` sub-class.

Tip: Interaction with the persistence layer is mediated through the `User` business object (note the `book.addUser(aUser)` method invocation). This helps keep the application logic decoupled from the Struts framework.

ActionForward

On completion, `perform` returns an `ActionForward` to the `ActionServlet` controller. In this manner, the `Action` object notifies the controller where control should next transfer to. In our example, control will always pass to the handler identified by the `ActionMapping` with key “success”. Configuration of `ActionMappings` is discussed in 13.6.2, “Registering `ActionMapping` and `ActionForward`” on page 179.

13.5 Logging using Log4j

The `AddUserAction` class utilizes `Log4j` to report error conditions; other classes use `Log4j` to trace program execution (see Example 13-15 on page 189). The complete `Log4j` reference manual can be found at:

<http://jakarta.apache.org/log4j/docs/manual.html>

We begin by introducing `Log4j` components:

Loggers	Loggers enable independent application component logging. They can be useful to selectively enable logging in different application subsystems (for instance, enabling in the model component while disabling controller logging).
Appenders	Appenders control log message destinations (console, log files, or sockets, for example).
Layouts	Layouts format log messages.

On startup, `Log4j` reads its configuration file to establish the appropriate `Loggers`, `Appender`, and `Layouts` (configuration is discussed in 13.5.2, “Configuring `Log4j`” on page 177). The address book uses a simple root `Logger` (control logging for the entire application).

13.5.1 Using Log4j

Example 13-6 illustrates Log4j usage in application code.

Example 13-6 Using Log4j

```
import org.apache.log4j.Logger;           1
public class Foo {
    static Logger log = Logger.getLogger(Foo.class);  2

    public Foo() {
        log.debug("this is a debug msg");           3
        log.info("this is an info msg");             4
        log.warn("this is a warning msg");           5
        log.error("this is a warning msg");           6
        log.fatal("this is a error msg");            7
    }
}
```

1. Import the Log4j package.
2. Obtain a Logger instance (note the static Logger getLogger method takes a Java Class parameter).
3. Invoke debug method to generate debugging messages.
4. Invoke info method to generate informational messages.
5. Invoke warn method to generate warning messages.
6. Invoke error method to generate error messages.
7. Invoke fatal method to generate error messages.

Each of the logging methods are overloaded to accept a second parameter of type Throwable (this allows stack trace to be included with the log message).

13.5.2 Configuring Log4j

By default, Log4j will read a configuration file named log4j.properties (illustrated in Example 13-7) located in the application root directory.

Example 13-7 Log4j configuration file (log4j.properties)

```
# Development logging
log4j.rootLogger=DEBUG, stdout           1

# Production logging
#log4j.rootLogger=INFO, stdfile, email    2

log4j.appender.stdout=org.apache.log4j.ConsoleAppender           3
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%r [%t] %p %c %x - %m%n
```

```
log4j.appender.stdfile=org.apache.log4j.RollingFileAppender
log4j.appender.stdfile.File=sg246807_stdout.log
log4j.appender.stdfile.MaxFileSize=100KB
log4j.appender.stdfile.MaxBackupIndex=1
log4j.appender.stdfile.layout=org.apache.log4j.PatternLayout
log4j.appender.stdfile.layout.ConversionPattern=%r [%t] %p %c %x - %m%n
```

4

```
log4j.appender.email=org.apache.log4j.net.SMTPAppender
log4j.appender.email.SMTPHost=localhost
log4j.appender.email.From=tomcat4@lnxa.appdev.net
log4j.appender.email.To=maintenance@lnxa.appdev.net
log4j.appender.email.Subject=SG246807 Exception
log4j.appender.email.layout=org.apache.log4j.PatternLayout
log4j.appender.stdfile.layout.ConversionPattern=%r [%t] %p %c %x - %m%n
```

5

1. For development, logging is enabled for message level **DEBUG** and higher. Messages are directed to Appender `stdout`.
2. For production, logging is enabled for message level **INFO** and higher. Messages are directed to Appenders `stdfile` and `email`.
3. Appender `stdout` is defined to be the console, it uses a standard Layout, and messages are formatted according to the defined `ConversionPattern` (see the reference manual for formatting pattern specifications).
4. Appender `stdfile` is defined to be the file `sg246807_stdout.log`. Additional parameters define log file size and backup frequency.
5. Appender `email` is defined to be an SMTP email message. Additional parameters define the recipient, sender, and subject.

13.6 Struts framework configuration

We now discuss how to configure the Struts framework.

13.6.1 Registering ActionForm beans

To register ActionForm beans, add a `<form-bean>` stanza to the `WEB-INF/struts-config.xml` file (in the `<form-beans>` section). Example 13-8 shows the stanza which registers `AddUserFormBean` (Java class `com.ibm.itso.sg246807.form.AddUserFormBean`) using the name `addUserForm`.

Example 13-8 ActionForm configuration in WEB-INF/struts-config.xml

```
<!-- ===== Form Bean Definitions ===== -->
<form-beans>
  <form-bean name="addUserForm"
    type="com.ibm.itso.sg246807.form.AddUserFormBean"/>
  <form-bean name="searchUserForm"
    type="com.ibm.itso.sg246807.form.SearchUserFormBean"/>
</form-beans>
```

13.6.2 Registering ActionMapping and ActionForward

To register ActionMappings, add an `<action>` stanza to the `WEB-INF/struts-config.xml` file (in the `<action>` section). Associated with an Action are ActionForwards (identified by a `<forward>` stanza). Example 13-9 shows the Action configuration for `AddUserAction`.

Example 13-9 ActionMapping configuration in WEB-INF/struts-config.xml

```
<!-- ===== Action Mapping Definitions ===== -->
<action-mappings>
  <action path="/adduser"
    type="com.ibm.itso.sg246807.action.AddUserAction"
    name="addUserForm"
    scope="request"
    input="/useradmin/addUser.jsp"
    validate="true">
    <forward name="success" path="/useradmin/addUser.jsp"/>
  </action>
```

Note: Note the association of the `AddUserFormBean` to the `AddUserAction` (using the name `addUserForm` defined in the `<form-beans>` section). This bean is created in the `addUser.jsp` file (`input="/useradmin/addUser.jsp"`).

13.6.3 Configuring ActionServlet

As a final step, the `ActionServlet` must be configured, and the application server notified to direct HTTP requests to it.

Example 13-10 ActionServlet configuration in WEB-INF/web.xml

```
<!-- Action Servlet Configuration -->
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>application</param-name>
```

1

2

```

        <param-value>AppResources</param-value>
    </init-param>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>detail</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>validate</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<!-- Action Servlet Mapping -->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

3

4

1. Associates the Struts ActionServlet to name action.
2. Identifies the location of the application resource file.
3. Identifies the location of the Struts configuration file.
4. Maps URLs of the form *.do to the ActionServlet.

A complete description of all initialization parameters can be found at:

<http://jakarta.apache.org/struts/api/index.html>

See the javadoc section on ActionServlet.

13.7 The persistence layer

We now discuss adding persistence to the application. Relational databases are commonly used to store data. When accessing a relational database using Java, the JDBC interface is commonly used. It provides a standardized interface to a wide variety of database implementations (enabling the application code to

remain independent of that implementation). The complete JDBC reference can be found at:

<http://java.sun.com/products/jdbc/>

In addition to JDBC, the address application implements a *persistence layer* based on calls to a native C shared library. The shared library in turn executes static SQL against a DB2 database. These alternative implementations are represented in Figure 13-2.

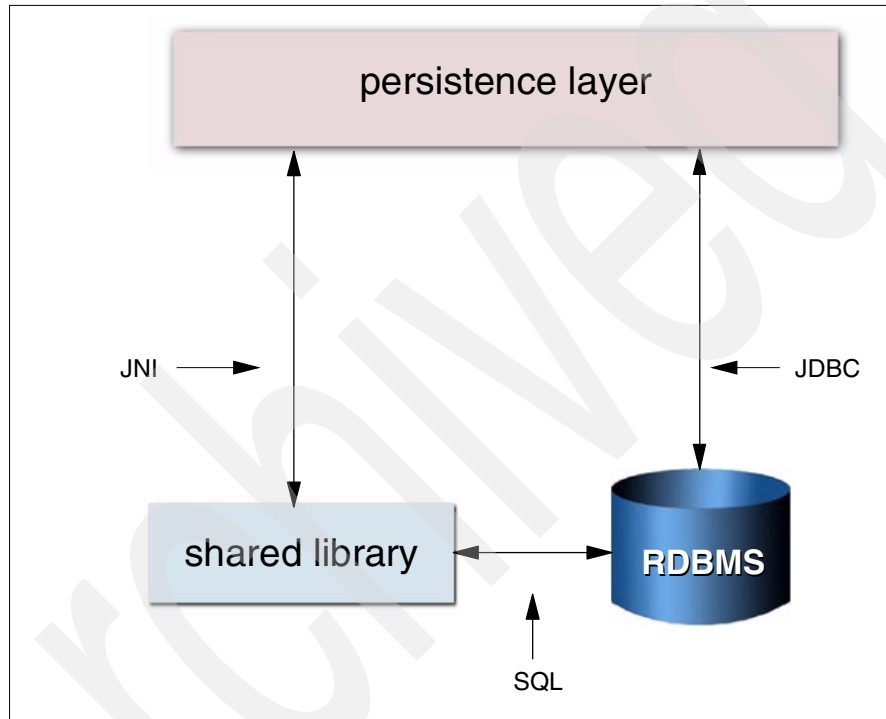


Figure 13-2 The persistence layer implementation

13.7.1 Data abstraction in the persistence layer

To isolate the persistence layer implementation from the application, we use an abstract Java class. Example 13-11 shows the Addressbook class the application uses to interface to the persistence layer.

Example 13-11 The abstract AddressBook persistent class (Addressbook.java)

```
package com.ibm.itso.sg246807.data;

import java.sql.SQLException;
```

```

import com.ibm.itso.sg246807.bo.User;
import com.ibm.itso.sg246807.ConfigurationManager;

public abstract class Addressbook {
    protected static Addressbook book;
    static {
        String connectionType =
            ConfigurationManager.getInstance().getProperty("connection.type");
        if (ConnectionManager.CON_NATIVE.equals(connectionType)) {
            book = new JniAddressbook();
        }
        if (ConnectionManager.CON_STUB.equals(connectionType)) {
            book = new StubAddressbook();
        }
        if (ConnectionManager.CON_POOL_STRUTS.equals(connectionType) ||
            ConnectionManager.CON_POOL.equals(connectionType) ||
            ConnectionManager.CON_SINGLE.equals(connectionType)) {
            book = new JdbcAddressbook();
        }
        if (book == null)
            throw (new IllegalStateException("Cannot instantiate" +
                                             " addressbook type " +
                                             connectionType));
    }
    protected Addressbook() {}
    public static Addressbook getInstance() throws SQLException {
        return book;
    }
    abstract public void addUser(User aUser) throws SQLException;
    abstract public java.util.Vector searchUser(User aUser)
        throws SQLException;
}

```

Note that the AddressBook expects subclasses to implement methods:

- addUser
- searchUser

13.8 The JDBC interface

In Example 13-12, we show the JDBC implementation for Addressbook - the JdbcAddressbook class (examining only the methods relevant to *add user*).

```
package com.ibm.itso.sg246807.data;

import java.util.Vector;
import java.sql.*;
import org.apache.log4j.Logger;
import com.ibm.itso.sg246807.bo.User;

public class JdbcAddressbook extends Addressbook {
    private static Logger log = Logger.getLogger(JdbcAddressbook.class);
    private ConnectionManager theConnectionManager;
    public JdbcAddressbook() {
        theConnectionManager = ConnectionManager.getInstance();
    }
    public void addUser(User aUser) throws SQLException {
        Connection con = null;
        try {
            con = theConnectionManager.getConnection();
            Statement stm = con.createStatement();
            StringBuffer query =
                new StringBuffer("insert into adrbook values ('")
                    .append(aUser.getFirstname())
                    .append(",")
                    .append(aUser.getLastname())
                    .append(",")
                    .append(aUser.getEmail())
                    .append(",")
                    .append(aUser.getPhone())
                    .append(")");
            stm.executeUpdate(query.toString());
            stm.close();
        } catch (SQLException ex) {
            theConnectionManager.rollback(con, ex);
            throw (new SQLException());
        } finally {
            theConnectionManager.closeConnection(con);
        }
    }
}
```

-
1. The JdbcAddressbook constructor obtains the single instance of the ConnectionManager class (used to manage connections to the database).
 2. A database connection is obtained from the ConnectionManager.
 3. An SQL statement handle (Statement) is obtained from the Connection.
 4. An SQL insert statement is constructed using values from the User instance.
 5. The insert statement is executed.

6. Once executed, the Statement is closed (freeing any allocated resources).
7. In the event of an error, the unit of work is rolled-back (using the ConnectionManager instance).
8. Finally, the Connection is released. This will either close the Connection entirely, or return it to a pool (available for subsequent requests).

Connection pooling is discussed further in the next section.

13.9 Connection pooling

When using a relational database, an application must first establish a connection to the database engine. Subsequent database operations utilize this connection. On completion, an application is responsible for closing the connection in order to free allocated resources (on both the server and the client). The process of establishing a database connection can take a substantial amount of time. (For operations that require one or two SQL operations, the time required to connect may exceed the time required to execute the SQL statements.)

Applications that wish to optimize performance should be aware of the overhead involved in establishing database connections, and if possible attempt to minimize the impact. One important technique is *connection pooling*, which means that on startup, an application opens a pool of connections. When servicing requests, rather than allocating a new connection (good only for the duration of request servicing), the application will instead obtain a connection from the available connections. On completion of the request, the connection is returned to the connection pool. On application shutdown, all connections will be released.

In our example, a separate class (named ConnectionManager) is responsible for establishing connections, and pooling those connections. In Example 13-13 we show the initialization of the ConnectionManager instance.

Example 13-13 Initializing a database connection - ConnectionManager.java

```
package com.ibm.itso.sg246807.data;
import java.sql.*;
import java.util.Properties;
import javax.sql.DataSource;
import org.apache.log4j.Logger;
import org.apache.struts.util.GenericDataSource;
import com.ibm.itso.sg246807.ConfigurationManager;

public class ConnectionManager {
    public final static String CON_SINGLE = "single";
```

```

public final static String CON_POOL_STRUTS = "pool_struts";
public final static String CON_NATIVE = "native";
private static ConnectionManager manager;
private static javax.sql.DataSource aDataSource;
private static GenericDataSource strutsDataSource;
private static Logger log = Logger.getLogger(ConnectionManager.class);
private static native void nativeConnect(String connectname);
private static native void nativeDisconnect();
private static String mConnectionType;
private static ConfigurationManager conf;
static {
    manager = new ConnectionManager();
    conf = ConfigurationManager.getInstance();
    mConnectionType = conf.getProperty("connection.type");
    if (CON_NATIVE.equals(mConnectionType)) {
        String NATIVE_CONNECT_NAME = conf.getProperty("native.url");
        System.loadLibrary("JniAddressbook");
        nativeConnect(NATIVE_CONNECT_NAME);
    }
    if (CON_SINGLE.equals(mConnectionType)) {
        String JDBC_DRIVER_CLASS = conf.getProperty("db.driverclass");
        try {
            Class.forName(JDBC_DRIVER_CLASS);
        } catch (ClassNotFoundException ex) {
            log.error(JDBC_DRIVER_CLASS + " not in classpath");
        }
    }
    if (CON_POOL_STRUTS.equals(mConnectionType)) {
        strutsDataSource = new GenericDataSource();
        strutsDataSource.setAutoCommit(true);
        strutsDataSource.setDescription("DB Connection Pool");
        strutsDataSource.setDriverClass(
            conf.getProperty("db.driverclass"));
        strutsDataSource.setMaxCount(4);
        strutsDataSource.setMinCount(1);
        strutsDataSource.setUser(conf.getProperty("db.user"));
        strutsDataSource.setPassword(conf.getProperty("db.pass"));
        strutsDataSource.setUrl(conf.getProperty("db.url"));
    }
}
private ConnectionManager() {}
public static ConnectionManager getInstance() { return manager; }
}

```

-
1. The single ConnectionManager instance is created as part the class initialization.

2. Connection configuration is read from a configuration file (see Example 13-14 for details on configuration). Connections are initialized based on those parameters.
3. A native connection (discussed in Section 13.10, “The Java Native Interface” on page 188) is requested.
 - a. The JNI shared library is loaded.
 - b. The JNI nativeConnect method is called to establish a connection.
4. A single, non-pooled connection type is requested. Using this connection type, a database will be established on each request and discarded at completion. The JDBC driver is loaded based on the configured driver name.
5. A pooled connection type is requested. Connections are reused across requests and are managed by Struts (using the `org.apache.struts.util.GenericDataSource` class). The `GenericDataSource` is configured using input parameters. Some values to note are:
 - a. Maximum database connections is set to 4.
 - b. Minimum connections is set to 1.

A complete description of all configuration parameters can be found at:

<http://jakarta.apache.org/struts/doc-1.0.2/api/index.html>

See the `GenericDataSource` class documentation.

13.9.1 Connection configuration

`ConnectionManager` is configured from the `sg246807.properties` file (shown in Example 13-14).

Example 13-14 Configuring connections (sg246807.properties)

```

# Define postfix for the property names
com.ibm.itso.sg246807.ENVIRONMENT=dev                                1

# Development
# Connection type: [native, single, pool_struts]
connection.type.dev=single                                           2
# JDBC connection
db.user.dev=db2inst1                                                3
db.pass.dev=*****
db.url.dev=jdbc:db2://10.1.1.136/sample
db.driverclass.dev=COM.ibm.db2.jdbc.net.DB2Driver
# Native connection
native.url.dev=sample:db2inst1:ibmdb2                                4

```

```
# Production
connection.type.prod=pool_struts
# JDBC connection
db.user.prod=db2inst1
db.pass.prod=*****
db.url.prod=jdbc:db2://10.1.1.136/ADRBOOK
db.driverclass.prod=COM.ibm.db2.jdbc.app.DB2Driver
# Native connection
native.url.prod=ADRBOOK:db2inst1:ibmdb2
```

1. Configuration parameters may be set for development (dev) and for production (prod) environments. Development values will be used.
2. Connection type is set to single, indicating no connection pooling will be used.
3. JDBC parameters are assigned. The parameter definitions are as follows:

db.user.dev	Userid used to connect to the DB2
db.pass.dev	Password to supply
db.url.dev	JDBC URL of the DB2 database
db.driverclass.dev	DB2 JDBC driver to use (discussed in detail in the next section)
4. JNI parameters are set.

DB2 JDBC Drivers

Two drivers are available with DB2: the JDBC net driver and the JDBC app driver.

Using the net driver, an application can access the DB2 server without installing a DB2 client. However, the **db2jstrt** command must be executed on the server machine. Figure 13-3 illustrates operation of the net driver. The net driver class name (db.driverclass.dev) is:

```
COM.ibm.db2.jdbc.net.DB2Driver
```

To use the app driver, a DB2 client must be installed. The app driver performance is better than the net driver, however. Figure 13-4 illustrates operation of the app driver. The app driver class name (db.driverclass.dev) is:

```
COM.ibm.db2.jdbc.app.DB2Driver
```

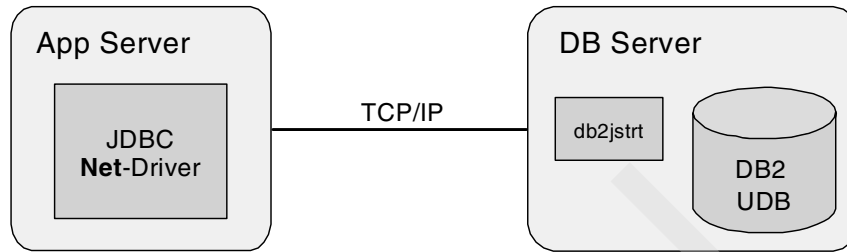


Figure 13-3 DB2 JDBC Net-Driver

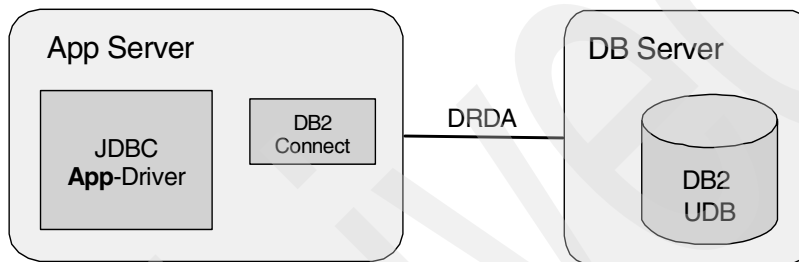


Figure 13-4 DB2 JDBC App-Driver

13.10 The Java Native Interface

The Java Native Interface (JNI) framework allows Java classes to call functions written in C or C++. In this section, we discuss a JNI interface which implements persistence in a relational database using shared libraries written in C. See 14.2, “Creating and using libraries” on page 196 for a discussion of the shared library implementation. For an overview the static SQL implementation, see 16.1, “Using embedded SQL in DB2 UDB applications” on page 238.

13.10.1 Using JNI in Java code

Use the following steps to incorporate native code in a Java program.

1. Begin by writing the Java program. Create a Java class that declares the native method; this class contains the declaration for the native method. It should also includes a static method which calls the native method.
2. Compile the Java program to create a Java class.

3. Generate a header file for the native method using the **javah** tool with the native interface flag **-jni**. Once you have generated the header file you have the formal signature for your native method.
4. Write the implementation of the native method.
5. Compile the header and the implementation files into a shared library file.

In our example, we show the Java class (`JniAddressBook` - a subclass of `AddressBook`) that invokes a number of native methods. These methods are all implemented in `JniAddressBook.c`.

Example 13-15 JniAddressbook persistence class (JniAddressbook.java - partial)

```
package com.ibm.itso.sg246807.data;

import java.sql.SQLException;
import com.ibm.itso.sg246807.bo.User;

public class JniAddressbook extends Addressbook {
    private native void nativeAddUser(String firstname,
                                     String lastname,
                                     String email,
                                     String phone);

    public void addUser(User aUser) throws SQLException {
        nativeAddUser(aUser.getFirstname(),
                     aUser.getLastname(),
                     aUser.getEmail(),
                     aUser.getPhone());
    }
}
```

13.10.2 Implementing the native code in C

The native C implementation is shown in Example 13-16.

Example 13-16 JniAddressbook native implementation (JniAddressbook.c - partial)

```
#include <jni.h>
#include "com_ibm_itso_sg246807_data_JniAddressbook.h"
#include "itsodb.h"

JNIEXPORT void JNICALL
Java_com_ibm_itso_sg246807_data_JniAddressbook_nativeAddUser(
    JNIEnv *env,
    jobject obj,
    jstring firstname,
    jstring lastname,
    jstring email,
```

1
2
3

4

```

        jstring phone) {
    const char *str1 = (*env)->GetStringUTFChars(env, firstname, 0);
    const char *str2 = (*env)->GetStringUTFChars(env, lastname, 0);
    const char *str3 = (*env)->GetStringUTFChars(env, email, 0);
    const char *str4 = (*env)->GetStringUTFChars(env, phone, 0);
    struct ITS0_user newUser;
    ITS0_init_user(&newUser, str1, str2, str3, str4);
    ITS0_addUser(&newUser);
    (*env)->ReleaseStringUTFChars(env, firstname, str1);
    (*env)->ReleaseStringUTFChars(env, lastname, str1);
    (*env)->ReleaseStringUTFChars(env, email, str1);
    (*env)->ReleaseStringUTFChars(env, phone, str1);
    return;
}

```

1. Include the `jni.h` to define Java data types.
2. Include the header file generated by **javah**. This file is named using the package and class name from Java implementation (`JniAddressbook.java`), replacing `'.'` characters with `'_'`.
3. Include the header file for the C shared library (defines `ITS0_init_user` and `ITS0_add_user` functions).
4. Define the native function prototype.
 - a. `JNIEXPORT void JNICALL` defines a void return value callable by Java.
 - b. `Java_com_ibm_itso_sg246807_data_JniAddressbook_nativeAddUser` defines the function name (Java + package name + class + method) assigned by **javah**.
 - c. `JNIEnv *` is the first parameter passed to the function; `jstring` is the datatype of a Java String.
5. `GetStringUTFChars` converts a Java String to C character array.
6. Every character array allocated by `GetStringUTFChars` should be freed by `ReleaseStringUTFChars`.

The complete JNI reference is available at:

<http://java.sun.com/products/jdk/1.2/docs/guide/jni/>

13.10.3 Building the JNI shared library

To create a shared library from the native C code, we use the **make** utility, supplying the Makefile shown in Example 13-17.

Example 13-17 Makefile to build JNI shared library

```
CC      =      gcc
INCS    =      -I /usr/local/itsodb/include \
               -I /opt/IBMJava2-s390-131/include      1

TARGET_DIR = ../../dist/lib                        2

CFLAGS  = -g -shared -L /usr/IBMdb2/V7.1/lib $(USRDEF)  3
LIBS    = -litsodb -ldb2                               4

all: libJniAddressbook.so                          5

clean:
    rm -f ${TARGET_DIR}/libJniAddressbook.so

libJniAddressbook.so: JniAddressbook.c
    mkdir -p $(TARGET_DIR)
    gcc $(CFLAGS) $(INCS) JniAddressbook.c $(LIBS) \
        -o $(TARGET_DIR)/libJniAddressbook.so
```

1. The `itso.h` file is found in the `/usr/local/itsodb/include` directory.
2. The JNI shared library will be written to the `../../dist/lib` directory.
3. The `$USRDEF` variable points to the directory location of the `libitsodb.so` library.
4. The `libitsodb.so` library implements the `ITS0_ SQL` functions.
5. The JNI shared library is named `libJniAddressbook.so`.

In 14.2, “Creating and using libraries” on page 196, we discuss how to build the `libitsodb.so` library, and 16.1.1, “Components of a DB2 UDB application” on page 238 discusses how to create static SQL.

We need to add a rule in the application `build.xml` file to invoke **make** from Ant. The rule is shown in Example 13-18.

Example 13-18 Ant rule to invoke make

```
<!-- Create shared lib's which have to be in the LD_LIBRARY_PATH -->
<target name="compile_c" depends="compile_java"
  if="native.support" description="Compile C part of application">
  <javah destdir="${src.c.dir}" classpath="${build.classes}"
    class="com.ibm.itso.sg246807.data.JniAddressbook"/>
  <javah destdir="${src.c.dir}" classpath="${build.classes}"
    class="com.ibm.itso.sg246807.data.ConnectionManager"/>
  <exec executable="make" dir="${src.c.dir}" failonerror="true"/>
</target>
```

Note: No predefined Ant rule exists to invoke **make**. We define a rule using the `<exec>` statement.

Shared libraries and more

In this chapter we discuss some practical aspects of developing applications on Linux for zSeries. We focus on:

- ▶ Creating and using libraries
- ▶ The `mmap` function
- ▶ Graphical user interface

The topics covered here are illustrated by the address book example.

14.1 Example overview

The address book is a simple application that allows users to add new entries or search for existing ones. We built the application from reusable components. Libraries described in this chapter also can be used in Java programs (see 13.10, “The Java Native Interface” on page 188). In this chapter the focus is on the C and C++ parts. We describe the library itself and two standalone programs.

14.1.1 Components of the address book example

Figure 14-1 illustrates the application structure.

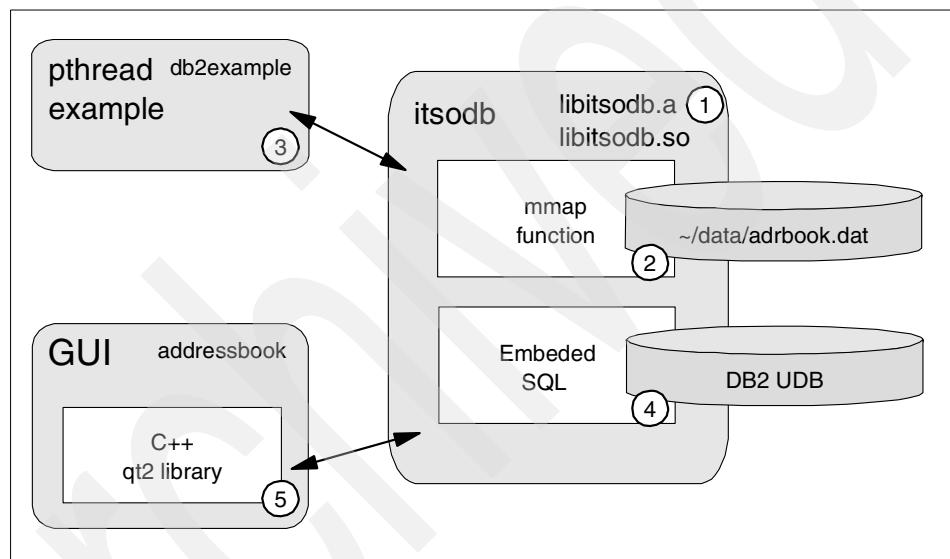


Figure 14-1 C and C++ part of the address book example

The address book example consists of the following parts:

1. Generic library wrapper (files `libitso.c`, `libitso.h`)
2. Poor man's database using flat files (`pmdb.c`, `pmdb.h`)
3. Embedded DB2 SQL (`sqldb.sqc`, `sqldb.h`)
4. Multithreaded test program
5. GUI interface

14.1.2 Implemented functionality

This section briefly describes features of the components.

The address book library

This is the main part of our example and it implements two sets of interfaces:

- ▶ **Poor man's database** which stores data in flat files (utilizing the `mmap` function).
- ▶ **SQL database** uses static DB2 SQL commands. In order to allow Java programs to use this library, we provide multithreaded access with connection pooling. We refer to this implementation as address book SQL database.

Each of the interfaces implements the following functions:

create (fname)	Creates a file-based database (does nothing in the SQL database).
open (fname)	Opens the database.
close ()	Closes the connection.
addUser (entry)	Adds a new entry.
initSearch (lname)	Initializes a search on last name. This function returns a handle that should be used in the retrieve function.
retrieve (handle,entry)	Places the next record of the search result into the entry field. Returns TRUE if the entry is a valid item and FALSE if all records have been retrieved already.

We use these functions with the `pmdb_` prefix when we refer to the poor man's database implementation, or `sql db_` when we refer to the embedded SQL version. For convenience, there are also macros with the `ITSO_` prefix which point to the first set of functions if `ITSO_PMDB` is defined, or to the second if not.

We show how we implement this library in the following sections:

- ▶ Section 14.2, "Creating and using libraries" on page 196
- ▶ Section 14.3, "A poor man's database" on page 207
- ▶ Section 15.2.6, "Synchronizing threads" on page 226
- ▶ Section 16.1, "Using embedded SQL in DB2 UDB applications" on page 238

The db2example program

This program dynamically loads the library and starts a given number of threads. Each of the tasks loads some data into the database in random periods.

In this example we focus on the following topics:

- ▶ Creating and starting threads within an application (additional details are in 15.2.1, “Using threads” on page 220)
- ▶ Loading shared objects on demand (see 14.2.8, “Dynamically linked libraries” on page 203 for details)

The Address book

GUI Addressable is a standalone application that can use both the poor man's database and the SQL database functions (depending on whether the `ITS0_PMDB` macro was defined during the compilation or not). By default, the `makefile` you can find in the source directory creates a DB2 version of the program. If you want to build the `mmap` example, invoke the `make` utility as follows:

```
make clean; make USRDEF=-DITS0_PMDB
```

In 14.4.1, “Graphical interface in a UNIX environment” on page 212 we describe how Qt library can be used to build a front-end for your application.

14.2 Creating and using libraries

Libraries are collections of precompiled functions and data structures that are ready to use in more than one program. There are three different ways of incorporating them into your application:

static libraries	Consist of compiled objects that are included in an executable program file at the time of compilation
shared libraries	Consist of compiled objects that are made available to programs at the beginning of execution. Images that reside in memory are shared between processes
dynamically linked libraries	Have the same format as a shared library but reload on an explicit request from the process

In the next two sections, we describe how to create and use the libraries.

14.2.1 Preparing object files

To prepare an object file for your library, you should compile but not link it. The `-c` option is essential. The following command produces the object file called `somefun.o` from the source file:

```
gcc -c somefun.c
```


In Example 14-1 we use Makefile rules:

Example 14-1 Makefile rules to compile C source

```
pmdb.o: pmdb.c pmdb.h itsodb.h
    $(CC) $(CCARG) -c $<

itsodb.o: itsodb.c itsodb.h
    $(CC) $(CCARG) -c $<
```

We suggest that you compile all objects for your library with the same set of a compiler options.

14.2.2 Inspecting object files

nm command

To check which symbols are implemented in an object file, use the **nm** command:

```
nm objectfile.o
```

Some of the more interesting information is related to the external symbols (uppercase values in the second column):

T	Function exported by the module
U	Function used by the module
C	Uninitialized global (non-static) variable
D	Initialized global (non-static) variable
R	Read-only data section

You can use these options:

-g	To see external symbols only
-u	To see undefined symbols only

This command works with static and shared libraries.

objdump command

The **objdump** command is useful for investigating the contents of the object file.

You can use the following options:

-d, -D	Display assembler contents of executable (-d) or all (-D) sections
-S	Display source code (if the file was compiled with -g option) mixed with disassembly
-t	Display the contents of the symbol tables

objdump displays a full list of operations when invoked with no parameters.

14.2.3 Static libraries

Static libraries are created by the `ar` program from a set of compiled object files. To produce a static library, we use the procedure described in the next section.

Note: Both types of libraries also can be created by the `libtool` program, which performs the same steps, but can apply options appropriate for your environment. Refer to the info system `binutils -> libtool` for details.

Preparing object files

Prepare an object file for your library by compiling but not linking it. The `-c` option is essential. The following command produces the object file called `somefun.o` from the source file:

```
gcc -c somefun.c
```

In our example we used these Makefile rules:

```
pmdb.o: pmdb.c pmdb.h itsodb.h
$(CC) $(CCARG) -c $<
itsodb.o: itsodb.c itsodb.h
$(CC) $(CCARG) -c $<
```

We suggest you compile all objects for your library with the same set of a compiler options.

Creating the library

You can create the whole library from all your files in one step:

```
ar csr libyourname.a somefun1.o somefun2.o ..
```

The same set of option allows you to add new files to an existing library:

```
ar csr libyourname.a newfun1.o
ar csr libyourname.a newfun2.o
```

The most useful options are the following:

- | | |
|----------|---|
| r | Insert files into the archive (and replace exiting ones with the same names). |
| x | Extract members of the archive. |
| t | Print the contents of the archive. |
| s | Update an objects index in the archive (this option is equivalent to running <code>ranlib</code> on the library). |
| d | Delete modules from the archive. |

Using static libraries

A static library is simply a collection of files included during the linking phase (see 1.2.4, “Compilation stages” on page 7). To include a static library in an executable, specify its:

Name	Using the <code>-l</code> option
Location	Using the <code>-L</code> option

Do not include the `lib` prefix or the `.a` extension in the name. If the static library resides in the `/usr/lib` or `/lib` directories, do not specify the `-L` option (these directories are in the linker’s standard search path).

To include the static library `/myproject/libitsodb.a`, link your executable as follows:

```
gcc -o myprog myprog.o objectfile.o -L/myproject/lib -litsodb
```

To include the static library `/usr/lib/libitsodb.a`, simply link as:

```
gcc -o myprog myprog.o objectfile.o -litsodb
```

When creating the executable, `gcc` searches for the `main` symbol and subsequently adds other references as needed. The search order for symbols is determined by the order specified on the command line (for both object and library files). The standard C libraries are searched last. This allows you to provide replacement implementations for standard functions. For example, to run your program with a different implementation of the `malloc` function, simply include the library or object file on the command line.

Tip: To track memory leaks, try using Electric Fence, a replacement for the standard `malloc` libraries (written by Bruce Perens). You can download it from:

<ftp://ftp.perens.com/pub/ElectricFence/>

14.2.4 Shared libraries

Shared libraries are loaded into memory at program execution time. As a consequence, an executable which refers to a shared library does not need to be recompiled when that library is changed, provided the interface has not changed (this provides backward compatibility).

Additionally, shared libraries conserve system memory: only one copy of the library text code is in memory. That copy is shared between all processes accessing that shared library.

Note: In execute mode, no process may overwrite the text section of a shared library.

All variables declared in a shared library are stored in process-local memory. This means each process maintains its own copy of shared library global variables and those variables cannot be modified by another process. (This may not be true of other non-UNIX operating systems.)

The process of loading shared libraries is quite complicated and can lead to a performance penalty. However, because the size of an executable can be reduced by using shared libraries and because the library may have been previously loaded, there can be a performance improvement. As a rule of thumb, shared libraries are best used when the cost of loading the library is low compared to the amount of time spent executing library code. For example, if your executable calls a library function only once, it may be better to use a static library instead (static library functions are included in the executable during the link phase and do not incur the cost of execution-time loading). If your executable makes repeated calls to library functions, it may be better to use shared libraries (the loading cost is low relative to execution time, the size of the executable may be smaller, and the shared library may already have been loaded into memory).

Before we describe how to create shared libraries, let's have a look at some usage considerations.

14.2.5 Using shared libraries

A shared library is not loaded by the application, but rather by the operating system *loader*. (This process depends on an executable format and is system-dependent. Shared libraries are loaded in different ways on Linux, AIX, and OS/390 UNIX). On Linux, the loader is named `ld-linux.so` and it relies on the ELF object format. The loader is a special library implicitly linked to every program that uses shared objects. When an executable using shared libraries starts, the loader is responsible for loading all required libraries.

Note: `ld-linux.so` has no `lib` prefix. It can be run as a standalone program in order to start an application with preloaded objects.

Shared library names

*Program-Library-HOWTO*¹ defines the categories of library names:

realname The name of the file that is produce by linker. Names follow the convention: *libname.so.major.minor.release*

where:

name is the name of the library

major is the version number (it should be changed whenever definition or behavior of exported symbols is changed)

minor is the lower version number (optional)

release is the release (optional).

soname The name used by the application in selecting which *realname* to load. Names follow the convention: *libname.so.major*

linkname The name specified at the linking stage (omitting the `lib` prefix and `.so` suffix). Names follow the convention: *libname.so*

The libraries are placed in the `lib` (`/usr/lib` or `/lib`) directory using their *realname*. The *soname* and *linkname* are symbolic links according to the convention:

- *soname* points to the proper implementation of the library version *major*.
- *linkname* points to this version of the library to be used during compilation.

Loader and ldconfig

When it is required to load a library, the loader consults (in this order) `/etc/ld.so.preload` and a cache stored in `/etc/ld.so.cache`. The cache is an index that tells the loader where a library can be found. It is provided as an efficiency mechanism. The cache is built using the **ldconfig** command:

```
ldconfig -v dir1 dir2 dir3 ...
```

The cache is created by examining the following:

- Command line parameters *dir1 dir2 dir3 ..*
- The file `/etc/ld.so.conf`
- Trusted directories `/usr/lib` and `/lib`.

When searching through directories, **ldconfig** not only appends the library path to the cache, but also creates a proper link to it (in *soname* style).

The *linkname* entry must be created explicitly on the library installation.

¹ Linux how-to files can be found in almost every distribution. You can also download files (in various formats: html, pdf, ps) from <http://www.tldp.org/docs.html>

Environment variables

Environment variables that change the behavior of the loader are:

LD_PATH_LIBRARY	A colon-separated list of directories where the loader searches for a library before searching the cache.
LD_PRELOAD	A colon-separated list of libraries (full or relative paths) to be loaded before the loading default libraries.
LD_DEBUG	Turns on tracking of the ld function. Set this to see all possible libraries to be loaded.

14.2.6 Building shared libraries

Prior to building a shared library, all functions to be included in an executable should be precompiled and stored in object files. To shorten the time required to load a shared library, an option to produce position-independent code (the **-fPIC** option) should be used when producing an executable. Do not use options like **-fomit-frame-pointer** if you want to preserve hints for a debugger.

Building a shared library is a straightforward process²:

```
gcc -shared -o libname.so.mj.mi.rl -Wl,-soname,libname.so.mj \
    somefun1.o somefun1.o -lusedlib1 -lusedlib2
```

Options specify:

- o** The output file name specified in *realname* format as described previously.
- Wl** Parameters passed to the linker, in this case, the *soname* parameter (commas will be replaced by spaces).

Note: We advise you to set *soname* (using the **-Wl** option in the preceding example) when creating shared libraries. When loading an application, the linker will load the shared library using *sonames*. If you do not specify *soname*, the loader will use the name specified during compilation (which may not be the shared library version intended by the application).

14.2.7 Investigating shared object dependencies

The **ldd** command prints the shared library names required by each program or shared library specified on the command line. All actual settings, like **LD_PRELOAD** or **LD_LIBRARY_PATH**, are taken into account.

```
$ldd ../db2driver/db2driver
libpthread.so.0 => /lib/libpthread.so.0 (0x40025000)
```

² If you plan to distribute your library on various platforms you may consider use of libtool.

```
libdl.so.2 => /lib/libdl.so.2 (0x4003b000)
libc.so.6 => /lib/libc.so.6 (0x40040000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Important: `ldd` internally executes the program and debugs its output. Do not execute `ldd` on programs of unknown origin.

14.2.8 Dynamically linked libraries

Dynamically linked libraries (DLL) are the same shared objects described previously. The difference is in when and how they are made available to a program. DLLs are not automatically loaded into memory by the loader. Instead, they are explicitly loaded by an application. It is the application's responsibility to resolve all symbols in the DLL it loads. The advantage of using DLLs is that the application can decide which module is to be loaded and when it should be loaded (determined at execution time). For instance, an application might read a configuration file to determine which library to load in order to implement a given functionality. This kind of library is often used to provide optional features that are unknown at compilation time. (For example, DB2 and MQSeries exits are implemented in this way.)

Now we describe the APIs you use to explicitly load libraries from your application. To use these functions add the following `include` directive to your source files:

```
#include <dlfcn.h>
```

and link your program with the `ld` library:

```
gcc -o myprog myprog.c -ldl
```

Dynamic loading of the libraries was used in our `db2driver` example (relevant fragments are presented in Example 14-2).

Example 14-2 Loading shared objects (db2driver.c)

```
01:  #include<dlfcn.h>
02:  ...
03:
04:  int (*sqldb_open_f)(char*);
05:  int (*sqldb_close_f)();
06:  int (*sqldb_addUser_f)(struct ITS0_user *);
07:  int (*sqldb_initSearch_f)(const char *);
08:  int (*sqldb_retrieve_f)(int, struct ITS0_user *);
09:
10:  main(int argc, char *argv[]){
11:
12:      int i;
```

```

13: void *lib=dlopen("libitsodb.so",RTLD_LAZY);
14: char *error;
15:
16: if(!lib){
17:     fprintf(stderr,"Error while opening library:%s\n", dlerror());
18:     exit(1);
19: }
20:
21: fprintf(stderr,"Resolving symbol :%s\n","sqldb_open");
22: sqldb_open_f = (int (*)(char *)) dlsym(lib, "sqldb_open");
23: if ((error = dlerror()) != NULL) {
24:     fprintf (stderr, "%s\n", error);
25:     exit(1);
26: }
27: /* main part of the program */
28: ...
29: dlclose(lib);
30: }

```

Opening libraries

To open a library file, use the following function:

```
void *dlopen (const char *filename, int flag);
```

If the filename does not begin with a "/", the following directories are searched:

- Colon-separated list in LD_LIBRARY_PATH environment variable
- Libraries cached in /etc/ld.so.cache
- /usr/lib
- /lib

Tip: The current executable file is opened if the filename is NULL.

The second parameter `flag` can be set to:

RTLD_LAZY	Resolve undefined symbols as code from the dynamic library is executed.
RTLD_NOW	Resolve all undefined symbols on <code>dlopen</code> call.
RTLD_GLOBAL	Makes the symbols from the library available to subsequently loaded ones. This flag is ORed with RTLD_NOW or RTLD_LAZY (for instance, RTLD_LAZY RTLD_GLOBAL)

If `dlopen` fails, it returns NULL.

Symbol lookup

Before using an object from a library, an executable must look for its symbol and store a reference to that symbol in a variable. Example 14-2 on page 203 shows how you can declare variables that point to functions (lines 04-09). The `dlsym` function searches the library for a symbol and returns its address (line 22 in our example).

```
void *dlsym(void *handle, char *symbol);
```

Releasing libraries

Use the `dlclose` function to inform that your application no longer needs the library. A reference count is maintained and the library is unloaded when the count goes to zero.

```
int dlclose (void *handle);
```

Checking for errors

If any of the functions described above fails, an error message is returned by `dlerror`.

```
const char *dlerror(void);
```

Note: This message is returned only once. Subsequent calls will return NULL. You should store the returned value in a variable, as shown in line 23.

14.2.9 Include files

When you create the library, all the symbols you would like to export should be declared in the header file supplied with the development version of the library. There are three versions of your library:

1. Source code distribution
2. Development distribution, which includes compiled libraries, header files, and some piece of documentation
3. Runtime, which is the library as shared object

The variables, functions, and data types that are exported should be declared as `extern` ones. If a function or a variable should not be exported, define it as `static`.

To prevent the compiler from including your header files multiple times, surround the header file with the following preprocessor statements:

```
#ifndef FILENAME_H
#define FILENAME_H
/* your definitions */
#endif
```

The C++ language allows a programmer to define more than one function with the same name provided that they use different data types for parameters or return values. Since the standard linkers distinguish symbols (functions being one type of symbol) only by name, a different naming convention is used by the C++ compiler. The C++ function names includes the data types separated by an underscore character (_).

If your library is pure C code (like the address book library described here), you should provide a proper header to use it in the C++ programs. All definitions should be surrounded with the following construct:

```
extern "C" {
/* your definitions */
}
```

A C++ compilation may be identified with a `__cplusplus` flag, so often header files look like the one shown in Figure 14-3.

Example 14-3 Header file template.

```
#ifndef FILENAME_H
#define FILENAME_H

#ifdef __cplusplus
extern "C" {
#endif

/*external declarations goes here*/

extern int mylibfun(int);
#ifdef __cplusplus
}
#endif

#endif
```

14.3 A poor man's database

The basic facilities used to store data in Linux are files. The most commonly used functions to manipulate file data are **write** and **read**. In this section, we describe a different approach. File contents are mapped to the process address space and manipulated in memory. This solution is used in a poor man's version of the address book database.

14.3.1 Memory mapped files

We use the **mmap** function to implement the file-based version of the address book library. This system call has several advantages over the read/write combo:

- ▶ The file contents are not buffered by the kernel.
- ▶ Only one copy of the data may reside in the memory. When considering the double buffering that occurs when running as a Linux guest under zVM, this can lead to a big performance improvement.

Example 14-2 illustrates a memory-mapped file accessed by two processes.

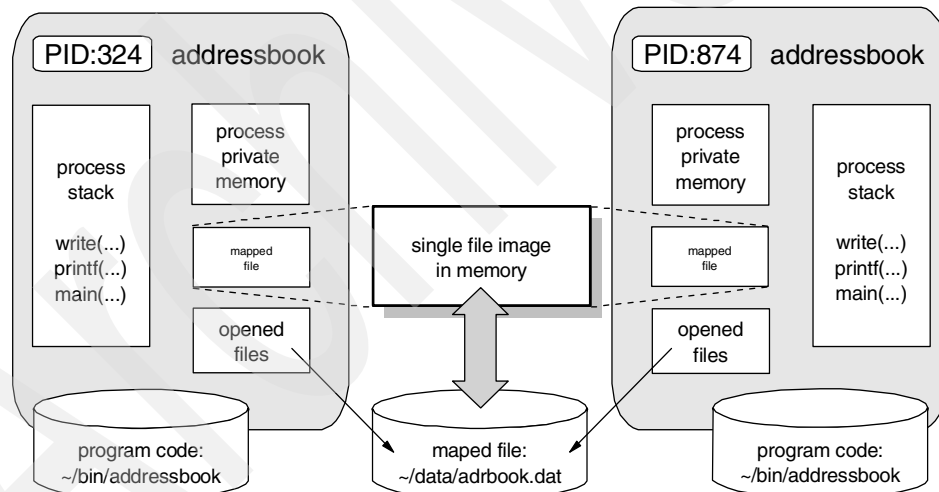


Figure 14-2 Mapping files in Linux processes

The **mmap** call looks as follows:

```
void *mmap(
    void *start,
    size_t length,
    int prot ,
    int flags,
    int fd,
    off_t offset);
```

Parameters are defined as:

start Specifies where within the process address space the file is to be mapped. This parameter is a hint only and often is set to 0. If you provide a value, remember to obey the Linux for zSeries addressing rules and align the address to the page size.

Note: You can get the page size for your architecture with the `getpagesize` call. In case of Linux on zSeries, it is 4096 bytes.

length Size of mapped data. You should allocate enough memory to accommodate the mapped file size before calling **mmap**. Otherwise the application may terminate with the SIGBUS signal.

prot Sets the memory protection access (must be compatible with the mode in which the mapped file is opened):

PROT_EXEC Pages may be executed.

PROT_READ Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE Pages may not be accessed.

flags At least one of the following two must be specified:

MAP_SHARED Share this mapping with all other processes that map this object. Writing to the region is equivalent to writing to the file. The file itself may not be see updates until the **msync** or **munmap** functions are called.

MAP_PRIVATE Create a private copy-on-write mapping. Writes to the region do not affect the original file.

Other flags may be specified (see the man page for **mmap**). One of the flags is **MAP_LOCKED**, which prevents the mapped data from being swapped.

Note: You can use also **mlock** to disable paging for the memory in the given range. See the **mlock** man page for details.

The sample code using **mmap** is presented in Example 14-4.

To close the mapped file, you should call **unmap** and then close the file as shown in lines 49-53.

```
int munmap(void *start, size_t length);
```

Parameters are defined as:

start	Specifies the beginning of the memory to unmap
length	Specifies the size of the memory to unmap

Example 14-4 Using mmap (pmdb.c - fragments)

```
01:  #include"stdio.h"
02:  #include"unistd.h"
03:  #include"errno.h"
04:  #include"sys/mman.h"
05:  #include"sys/types.h"
06:  #include"sys/stat.h"
07:  #include<fcntl.h>
08:
09:  struct pmdb_info{
10:      int entries;
11:  };
12:
13:  #define MAXENTRIES 1000
14:  #define MMAP_SIZE (sizeof(struct ITS0_user) \
15:                    *MAXENTRIES+sizeof(struct pmdb_info))
16:
17:  int mmap_fd;
18:
19:  static void *mmap_data;
20:  static struct pmdb_info *info;
21:  static struct ITS0_user *users;
22:
23:      /* ... */
24:
25:  int pmdb_open(char *fname){
26:
27:      mmap_fd = open(fname, O_RDWR );
28:
29:      CHECK_WITH_RETURN( (mmap_fd < 0), -1);
30:
31:      mmap_data=mmap( NULL,
```

```

32:         MMAP_SIZE,
33:         PROT_READ|PROT_WRITE,
34:         MAP_SHARED,
35:         mmap_fd,
36:         0);
37:
38:     CHECK_WITH_RETURN( (mmap_data == MAP_FAILED) , -1);
39:
40:     info = (struct pmdb_info *) mmap_data;
41:     users = (struct ITS0_user *) (mmap_data + sizeof(struct pmdb_info));
42:
43: }
44:
45: /* ... */
46:
47: int pmdb_close(char *fname){
48:
49:     if(mmap_data > 0)
50:         munmap(mmap_data, MMAP_SIZE);
51:
52:     if(mmap_fd >= 0)
53:         close(mmap_fd);
54:
55:     mmap_fd=0;
56:     mmap_data=0;
57: }
58:
59: /* ... */

```

14.3.2 Synchronizing memory and disk storage

The **msync** function writes changes to the memory image back to disk. Without this call, there is no guarantee that changes will be preserved.

msync syntax

The **msync** function has the following syntax:

```
int msync(const void *start, size_t length, int flags);
```

Parameters are:

start	Beginning of the memory to be written to disk. This address must be aligned to the page size.
length	Amount of data to be written.

flags Must be one of the following:

MS_ASYNC	Specifies that an update is to be scheduled, but the call returns immediately
MS_SYNC	Asks for an update and waits for it to complete.

Optional flags which may be ORed include:

MS_INVALIDATE	Invalidates other mappings of the same file (causing those mapping to be refreshed with the current file contents).
---------------	---

Logging changes

You can use this function to log changes in exactly the same way that real databases do. A commonly used technique is to manage data in two files: a snapshot and a log. On restart, the data is restored by replaying the snapshot and the log. When the log grows to a specified length, the application writes a new snapshot and empties the log. Log changes should be written synchronously.

14.4 Graphical user interface

In this section, we briefly describe how the address book standalone version was built. A sample screen from the application is in Figure 14-3.

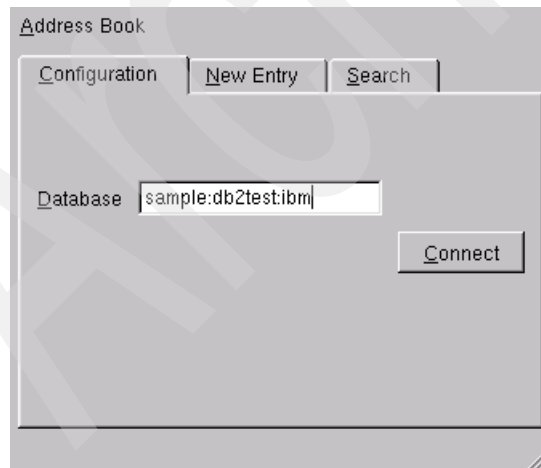


Figure 14-3 Address book application

The connection string is defined as:

- ▶ In the SQL version, it is defined as *database:userid:password*
- ▶ In the MMAP version, it is a path to the mapped file.

14.4.1 Graphical interface in a UNIX environment

At first sight, the X Windows concept may be a bit confusing. To clarify it, we begin by defining some relevant terms.

X-server A server that displays output from an X-client on a graphic terminal. It accepts user input from devices such as a keyboard or mouse and transmits those commands to the X-client.

X-client An application that runs (either locally or remotely) under user control from the X-server.

X-server communicates to X-clients using the X-Protocol. X applications running on Linux for zSeries act as X-clients. To run an X application on Linux for zSeries, an X-server is required to be running on the user machine (that machine must also have X-supported graphic hardware). In “Using XFree86 as an X Server under Cygwin” on page 292, we discuss one X-server available for Windows-based machines.

Figure 14-4 presents a simple scenario running the address book application.

The address book application is started on the Linux for zSeries machine. It reads the environmental variable `DISPLAY` and establishes a connection to the X server. The value of the `DISPLAY` variable is set as follows:

```
export DISPLAY=X_server_IP_address:X_server_number.screen_number
```

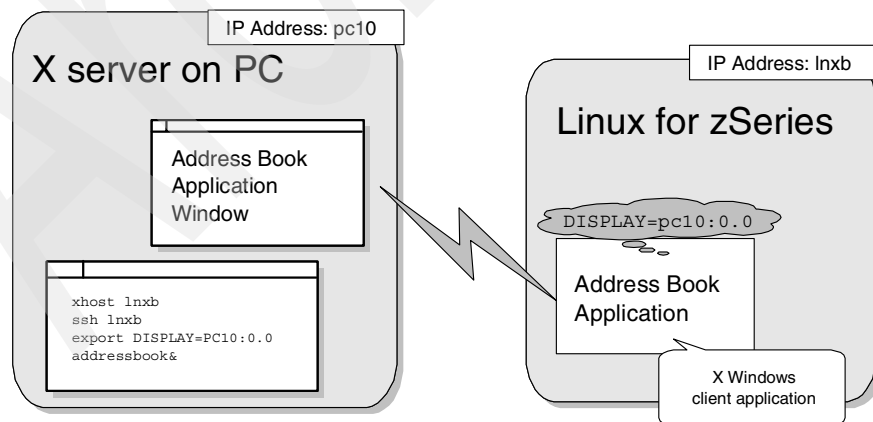


Figure 14-4 X windows session

The server address may be empty if your X-server runs on the same machine as the X-client. It happens when you work with X windows on your PC and run applications locally.

Note: The X-server number is usually 0 (zero) since there is only one X server running. You can start the second server on your PC with the X command:

```
X :1
```

Add access to it from a remote machine by:

```
export DISPLAY=mypc.appdev.net:1
```

If you switch to the first server with Alt+Ctrl+F7, the second one is available under Alt+Ctrl+F8.

The X server must be authorized before an X Windows client can connect to it. To authorize your server, issue the following command from any X terminal window:

```
xhost IP_address
```

A typical session might look as follows (assuming you work with Linux on a PC in an X Windows environment):

1. Authorize your server:

```
xhost 1nxb.appdev.net
```

2. Telnet to the host or issue the **ssh** command:

```
ssh 1nxb.appdev.net
```

3. On the server side, set the **DISPLAY** variable:

```
export DISPLAY=pc10.appdev.net:0.0
```

4. Start the application:

```
addressbook&
```

14.4.2 Qt library

Qt is a multi-platform, C++ library that allows you to write applications that run on several platforms (including Linux, BSD, and Windows). The most noticeable features in Qt are *signal* and *slots*. You connect them together to link an action to be performed with the corresponding objects. For example, a user's click on an *Add* button should trigger an `addEntry()` method, as shown in Example 14-5.

Example 14-5 Connecting signals and slots

In *AddressBook.h* file:

```
protected slots:
    void addEntry();
```

In *AddressBook.cc* file:

```
add = new QPushButton( "Add", input );
add->resize( add->sizeHint() );
grid->addWidget( add, 3, 2 );
connect( add, SIGNAL( clicked() ), this, SLOT( addEntry() ) );
```

We do not describe the library any further since the comprehensive Qt tutorial is available on the Internet at:

<http://doc.trolltech.com/3.0/index.html>

In the next section we focus on the build process.

The moc preprocessor and Makefiles

The extended syntax of Qt programs requires special preprocessing before the compilation. Whenever you declare signals or slots in your classes you should invoke the moc precompiler and include its output in your C++ files.

We use slots and signals in two modules: *AddressBook* and *MainWindow*.

Example 14-6 shows how the whole application is built with Makefile.

Example 14-6 Address book Makefile

```
#
# Makefile (by default) creates DB2 version of an address book example
# if you want to get a mmap example invoke the make utility as follows:
#
# make clean; make USRDEF=-DITSO_PMDB
#

CC      = g++
INCS    = -I/usr/lib/qt2/include -I/usr/local/itsodb/include
LIBS    = -litsodb -lqt
SRCS    = AddressBook.cc main.cc MainWindow.cc
OBSJS   = AddressBook.o main.o MainWindow.o

CFLAGS  = -g -L /usr/lib/qt2/lib -L $(USRDEF)

all: addressbook

clean:
    rm -f *.o addressbook
    rm -f *~
```

```

rm -f *.moc

addressbook: $(OBJJS)
gcc $(CFLAGS) $(LFLAGS) -o addressbook $(OBJJS) $(LIBS)

AddressBook.o: AddressBook.cc AddressBook.h AddressBook.moc
$(CC) $(CFLAGS) $(INCS) -c $<

MainWindow.o: MainWindow.cc MainWindow.h MainWindow.moc
$(CC) $(CFLAGS) $(INCS) -c $<

main.o: main.cc MainWindow.h
$(CC) $(CFLAGS) $(INCS) -c $<

MainWindow.moc: MainWindow.h
moc -o MainWindow.moc MainWindow.h

AddressBook.moc: AddressBook.h
moc -o AddressBook.moc AddressBook.h

depend:
    makedepend $(INCS) $(SRCS)

# DO NOT DELETE
#make depend entries goes here

```

The AddressBook and MainWindow objects depend not only on *.cc and *.h but on the corresponding *.moc file.

The moc file is included in the *.cc source file like a regular header. Here are the first few lines of the AddressBook.cc file:

```

#include "AddressBook.h"
#include "AddressBook.moc"

```




Designing for concurrent access

Presently, standard UNIX systems offers two levels of concurrency: processes and threads. Linux implementation of threads conforms to the POSIX standard, commonly called *pthreads*. In this chapter we describe:

- ▶ How pthreads can be used in applications
- ▶ How both processes and threads can be synchronized

15.1 UNIX processes

When a new program is started, Linux creates a process in which this program will be executed. Every process has its own private memory area, program stack, and collection of open files or other assigned resources. If data is to be exchanged with another process, the operating system provides the following mechanisms:

- Pipes
- Shared memory (IPC or MMAP)
- Files
- IPC message queues

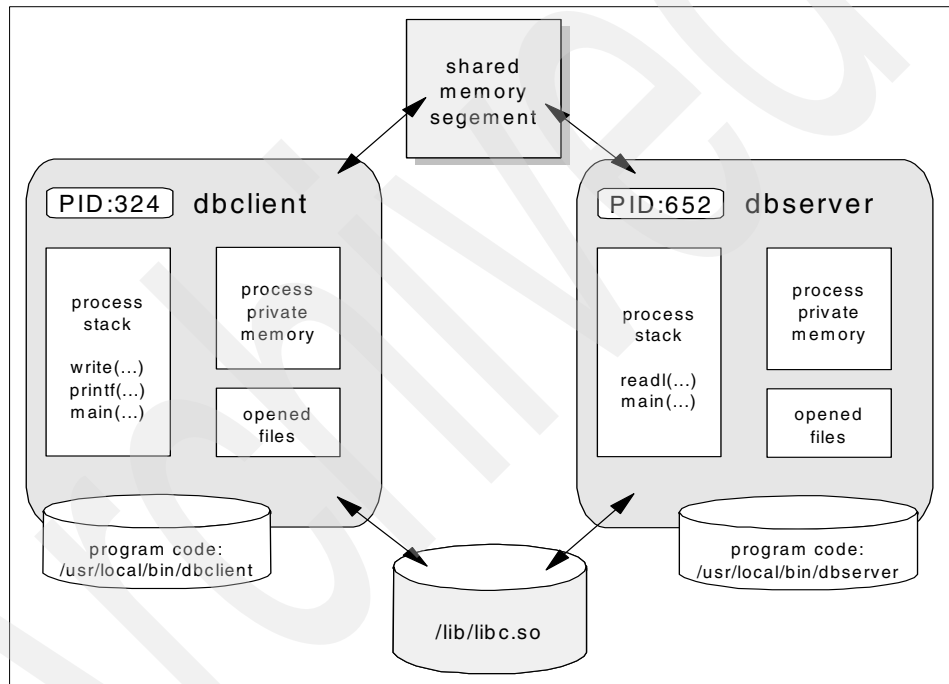


Figure 15-1 Processes in a UNIX environment

Processes are distinguished by PID numbers. Linux implements the `/proc` files system to obtain information on running processes.

Note that program code can be shared between processes using shared libraries (although not shown in this example, this is discussed in 14.2.4, “Shared libraries” on page 199). You can check what data and code segments are visible in the process by issuing:

```
cat /proc/pid/maps
```

A new process is created by cloning an existing one with the `fork` function. Usually, the `exec` function is then called to execute a new program. This procedure is shown in Example 15-1.

Example 15-1 Creating a new process with `fork`

```
switch( pid=fork() ){
case -1:
    /* handle an error */
    break;
case 0:
    /* child process, often the 'exec' function is called here */
default:
    /* parent process,
       value stored in pid equals PID of the child process */
}
```

The `fork` function returns to both the parent and the child process. You can distinguish these cases with the return value from `fork`:

0	Returned to the child process
pid	Process ID of the child is returned to the parent
-1	Returned to the parent in event of an error

Linux also has a system call named `clone` which allows the child process to share parts of its execution context with its parent process (such as the memory space, the table of file descriptors, and the table of signal handlers). For more details, see the `clone` manual page.

15.2 The `pthread`s library

Unlike a new process, a new thread executes in the address space where it was created. It has its own set of registers and stack, but its memory and process resources are shared. Unlike many other operating systems, there is no special ID for threads. An example of how PID numbers are assigned to threads is shown in Figure 15-2. PPID refers to the parent's PID.

Creating a new process is an expensive task. One benchmark that compares operating system performance measures how many forks can be executed in a given period of time. A new process uses system memory (impacting overall system performance).

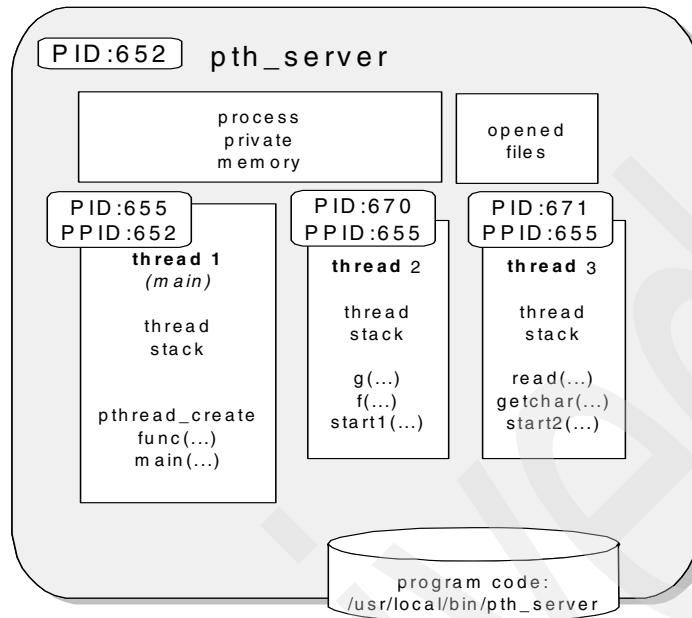


Figure 15-2 Multiple threads within an application

15.2.1 Using threads

In this section we describe how to start and terminate threads. Consider the code shown in Example 15-2.

Example 15-2 db2example.c (parts only)

```
01: #include<pthread.h>
02:
03: /* ... */
04:
05: #define NO_THREADS 10
06:
07: struct ctx{
08:     pthread_t pthread;
09:     int no;
10: } ctx[NO_THREADS];
11:
12:
13:
14: void *start_task(void *data){
15:
16:     struct ctx *myctx=(struct ctx *)data;
17:
```



```

18:  /* ... */
19: }
20:
21: main(int argc, char *argv[]){
22:
23:     int i;
24:
25:     /* ..... */
26:
27:     for(i=0; i < NO_THREADS; i++){
28:         ctx[i].no=i;
29:         pthread_create(&ctx[i].pthread, NULL,
30:             &start_task, (void*)&ctx[i]);
31:     }
32:
33:     for(i=0; i < NO_THREADS; i++){
34:         printf("Waiting for a thread %c.\n", 'A'+i);
35:         pthread_join(ctx[i].pthread, NULL);
36:         printf("Thread %c. ended\n", 'A'+i);
37:     }
38:
39:     /* ..... */
40: }

```

In this example, we prepare an array of thread contexts in lines 05-10 (struct `ctx`) in order to run many symmetric threads. Each entry contains the thread identifier (`pthread_t`) and the single parameter to be passed to the thread parameter (`no`). This approach may be useful when you want to store some thread private data in the data section of the main thread (instead on each thread stack). Moreover, these variables may be shared between threads as necessary. We pass a pointer to a struct `ctx` entry for each newly created thread in the function `pthread_create` (line 29).

Note: To use pthreads, pass the `-lpthread` option to the linker.

15.2.2 Creating threads

New threads are created with the function **`pthread_create`**:

```

int pthread_create(pthread_t *thread,
                  pthread_attr_t *attr,
                  void * (*start_routine)(void *),
                  void * arg);

```

Parameters are:

thread	Pointer to a variable used to identify the newly created thread.
attr	Pointer to the data structure called thread attribute object. See 15.2.4, “Thread attributes” on page 224 for further details, and the <code>pthread_attr_init</code> man page for a complete list of thread attributes.
start_routine	Pointer to the function at which the new thread starts execution.
arg	Pointer to a parameter to be passed to the function.

15.2.3 Thread termination

Threads end when:

- ▶ The thread completes execution.
- ▶ The thread calls the `pthread_exit` function:

```
void pthread_exit(void *retval)
```
- ▶ The thread was canceled with `pthread_cancel` call.

Note: The standard system call `exit` ends the whole process, including threads.

Another thread may be suspended until the thread ends. We used the function `pthread_join` in line 35 for this purpose:

```
int pthread_join(pthread_t th, void **thread_return);
```

The meanings of the parameters are:

th	Identifier of the thread to be waited on.
thread_return	Pointer to the return value from the joined thread. If not null, the value contains the return value of the thread. If equal to <code>PTHREAD_CANCELED</code> , the thread was cancelled.

Thread cancellation

This mechanism allows one thread to terminate another. Be aware that the thread that is to be canceled may have allocated resources and should be allowed to release them. There are two facilities that may be helpful:

- ▶ Cancellation points, which determine when termination may occur (to assure no resources will be left unallocated).

- The cleanup stack, a collection of the procedures to be called when the thread terminates (also when `pthread_exit` is invoked). Use `pthread_cleanup_push` to register a function or `pthread_clean_pop` to remove a function from the cleanup stack. When the thread terminates, routines are executed in the reverse order they were registered.

The behavior of threads on cancellation is governed by both the thread's *cancellation state* and *cancellation type* - set respectively by the functions:

- `pthread_setcancelstate`
- `pthread_setcanceltype`

Table 15-1 summarizes thread cancellation behavior based on *cancellation state* and *cancellation type*.

Table 15-1 *Setting thread cancellation behavior*

Behavior	Cancellation State	Cancellation Type
deferred	PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_DEFERRED
asynchronous	PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_ASYNCHRONOUS
disabled	PTHREAD_CANCEL_DISABLE	-

The default cancellation behavior is deferred. The behavior on cancellation is summarized as follows:

- deferred** Cancellation is deferred until a cancellation point is reached.
- asynchronous** Cancellation occurs immediately.
- disabled** The thread cannot be cancelled.

Cancellation points are classified as:

- Automatic
Cancellation occurs during a call to functions:
 - `pthread_cond_wait`
 - `pthread_cond_timewait`
 - `pthread_join`
- User-defined
Cancellation occurs during a call to function:
 - `pthread_testcancel`

15.2.4 Thread attributes

The pthreads standard defines attributes that allow you to specify thread properties. Linux for zSeries implements the following settings:

detachstate	Specifies the thread as:	
	PTHREAD_CREATE_JOINABLE	Thread can be joined (using <code>pthread_join</code>)
	PTHREAD_CREATE_DETACHED	Thread is detached (cannot be joined)
schedpolicy	Specifies the scheduling policy:	
	SCHED_OTHER	Regular, non-real-time
	SCHED_RR	Real-time, round-robin
	SCHED_FIFO	Real-time, first-in first-out
schedparam	Contains the scheduling parameters (essentially, the scheduling priority) for the thread.	
inheritsched	Determines how the scheduling policy and scheduling parameters are set:	
	PTHREAD_EXPLICIT_SCHED	Using the schedpolicy and schedparam attributes
	PTHREAD_INHERIT_SCHED	Inherited from the parent thread
scope	Defines the scheduling contention scope.	

Note: The only value supported in the LinuxThreads implementation is `PTHREAD_SCOPE_SYSTEM`, meaning that the threads contend for CPU time with all processes running on the machine.

stacksize	Sets the private stack size. The minimum value is <code>THREAD_STACK_MIN</code> (16384).
stackaddr	Sets the address for the thread stack. The address must follow addressing rules of Linux and zSeries architecture.

Note: Thread attributes can only be set when a thread is created.

15.2.5 Setting thread stack size

In this example, we show how the thread attribute **stacksize** can be set and the impact of this attribute on the program. Compile the program shown in Example 15-3 on page 225 (use default settings) and run it:

```
$ gcc -o thread thread.c -lpthread
$ ./thread 1000 24000
Segmentation fault
```

The thread stack size is too small. Increasing the stack to 105000 (1000*104 plus some extra space), we see:

```
$ ./thread 1000 105000
Done
```

Now we examine the optimization. Trying this example with a slightly greater number of calls, we see:

```
$ ./thread 1010 105000
Segmentation fault
```

When the compiler optimizes this code, it can get rid of the frame pointer. This saves some space on the thread stack.

```
$ gcc -O2 -o thread thread.c -lpthread
$ ./thread 1090 105000
Done
```

Note: Results shown vary according to architecture. This program will not be optimized in the same way on an i386 platform.

Example 15-3 Setting stack size for a thread

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
#include<bits/local_lim.h>

#ifdef _POSIX_THREAD_ATTR_STACKSIZE
#error No stack size
#endif
#ifdef _POSIX_THREAD_ATTR_STACKADDR
#error No stack addr
#endif
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING
#error No scheduling
#endif

pthread_t  pthread;
pthread_attr_t  pattr;

void f(int i){
    if(i != 0)
        f(i-1);
}
```

```

void *start_task(void *data){
    int i=*(int*)data;
    f(i);
    printf("Done\n");
}

main(int argc, char *argv[]){
    int i;
    int ssize;

    if(argc != 3 || (i=atoi(argv[1])) <= 0 || (ssize=atoi(argv[2])) <= 0){
        fprintf(stderr,"Usage:\n %s <count> <stack size>\n",argv[0]);
        exit(1);
    }

    pthread_attr_init(&pattr);
    if(pthread_attr_setstacksize(&pattr, ssize) != 0){
        fprintf(stderr,"Invalid stack size ! \n",argv[0]);
        exit(1);
    }
    pthread_create(&pthread, &pattr, &start_task,&i);
    pthread_join(pthread,NULL);
}

```

15.2.6 Synchronizing threads

In this section, we describe two pthreads objects: *mutexes* and *conditional variables*. We use these objects to prevent concurrent thread access to crucial data in the address book library. The relevant fragments of the code are presented in Example 15-4.

Example 15-4 Synchronizing access to data

```

01: #include<pthread.h>
02: /* ... */
03:
04: int ctx_free = NUM_CTX;
05: int ctx_close_connections = FALSE;
06:
07: pthread_mutex_t ctx_mutex;
08: pthread_cond_t  ctx_cond;
09:
10: sqldb_open(char *conname){
11:
12:     pthread_mutex_init(&ctx_mutex, NULL);
13:     pthread_cond_init(&ctx_cond, NULL);
14: /* ... */

```

```

15: }
16:
17: int assign_ctx(int action){
18:
19:     pthread_mutex_lock(&ctx_mutex);
20:
21:     while(ctx_free <= 0 )
22:         pthread_cond_wait(&ctx_cond,&ctx_mutex);
23:
24:     /* work with database contextes; ctx_free-- */
25:
26:     pthread_mutex_unlock(&ctx_mutex);
27:     return 0;
28: }
29:
30: int release_ctx(int ctx){
31:
32:     pthread_mutex_lock(&ctx_mutex);
33:
34:     /* work with database contextes; ctx_free++ */
35:
36:     pthread_cond_signal(&ctx_cond);
37:     pthread_mutex_unlock(&ctx_mutex);
38:     return 0;
39: }

```

15.2.7 Mutexes

A *mutex* is a variable that exists in one of two states:

unlocked	Not owned by any thread
locked	Owned by exactly one thread

A mutex can never be acquired by two different threads simultaneously.

Initializing a mutex

Mutexes are created and initialized using the function:

```

int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *mutexattr);

```

See line 12 in Example 15-4 for an example. In LinuxThreads, mutex attributes (the `pthread_mutexattr` data type) can assume values:

<code>PTHREAD_MUTEX_FAST_NP</code>	No checking, suspends the thread forever
<code>PTHREAD_MUTEX_RECURSIVE_NP</code>	Allows the thread to acquire the same mutex
<code>PTHREAD_MUTEX_ERRORCHECK_NP</code>	Results in an error on such attempts

Tip: We recommend setting `mutexattr` to `NULL` (take the default settings).

Note: This parameter is not portable.

Acquiring a mutex

To obtain a mutex, use:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Line 19 in Example 15-4 contains an example. This function locks the given mutex if the mutex is currently unlocked. Otherwise, the thread is suspended until the mutex is unlocked.

To conditionally obtain a mutex, use:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

This acts like `pthread_mutex_lock`. However, if the mutex is already locked, it ends immediately and returns `EBUSY`.

Releasing a mutex

To release a mutex, use:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Line 37 in Example 15-4 contains an example.

Destroying a mutex

Once all mutex operations are completed, use:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

to free resources associated with the mutex.

15.2.8 Conditional variables

Conditional variables allow threads to suspend execution and wait until some predicate on shared data is satisfied. We use a conditional variable in our example to wait until one of the database connection become available.

Declaring conditional variables

To declare a conditional variable, use

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *cond_attr);
```


See line 13 in Example 15-4 for an example.

Note: The current LinuxThreads implementation does not support attributes for conditions; the `cond_attr` parameter is ignored. Set it to `NULL`.

Waiting for conditions to occur

To temporarily release a mutex lock and wait on a conditional variable, use:

```
int pthread_cond_wait(pthread_cond_t *cond,
                     pthread_mutex_t *mutex);
```

This atomically unlocks the mutex (like the `pthread_unlock_mutex` function) and waits for the condition variable `cond` to be signaled. For example usage, see line 22 in Example 15-4.

To provide a time-out when waiting for the condition to occur, use:

```
int pthread_cond_wait(pthread_cond_t *cond,
                     pthread_mutex_t *mutex,
                     pthread_cond_timewait wait);
```

Signalling conditions

To pass a signal to other threads waiting on a condition, use:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

This restarts one of the threads that are waiting on the condition variable `cond`. If no threads are waiting on `cond`, nothing happens. If several threads are waiting on `cond`, exactly one is restarted (which one is not specified, however thread scheduling priorities play into the calculation). See line 36 in Example 15-4 for an example.

To signal all threads waiting on a condition, use:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

This restarts all threads that are waiting on `cond`. However, only one will acquire the mutex lock. Other threads are stopped and will wait their turn to acquire the mutex. The chosen thread may change the predicate and force other threads to wait once again. We suggest using a loop when `pthread_cond_wait` is called, as shown on line 2 in Example 15-4.

15.3 Controlling concurrent access

In this section we describe different ways of synchronizing access.

15.3.1 Locking using files

File locking can be used to protect resources from concurrent access by multiple processes. There are three standards forms of file locking:

- ▶ `lockf` for System V
- ▶ `flock` for BSD4.3
- ▶ `fcntl` for POSIX

Linux conforms to all three; however, `lockf` and `flock` are implemented by interfaces to `fcntl`. If any thread tries to get a lock on the file owned by the other thread, the function succeeds provided both threads are running within the same process.

15.3.2 IPC semaphores

A semaphore is a counter used to synchronize access to a shared data object. When processing a request to acquire a semaphore, the operating system will:

1. Test the usage count of the semaphore.
2. If the value is positive, the resource is available. The operating system grants the process access to the resource (allowing it to continue execution) and decrements the usage count by 1.
3. If the value is zero, the resource is unavailable. The operating system denies the process access to the process (putting it to sleep).

When a process releases a semaphore, the operating system will:

1. Increment the usage count of the resource.
2. Wake up any processes waiting on the resource and allow them to attempt to acquire the resource as outlined previously. Only one will actually acquire the resource (assuming only one process released the resource).

Semaphores have more functionality. Refer to the `semget`, `semctl`, and `semop` man pages for a detailed description.

A process must explicitly destroy a semaphore before it is removed from the system. Errant processes can sometimes leave residual semaphores on the system. To manually list and remove these residual semaphores, use:

```
ipcs -s    Lists allocated semaphores
```

ipcrm sem Removes specified resources.

Important: Semaphores cannot be used to synchronize threads within a process. A waiting thread will not be notified when the semaphore value is increased by another thread within the same process.

15.3.3 Pthread resources

The pthread library features described in 15.2.6, “Synchronizing threads” on page 226 work for threads operating within a single process. However, these features cannot be used to synchronize threads across processes. An attempt to do so will result in an error. To synchronize threads across processes, use *spinlocks*.

Spinlocks

Spinlocks are synchronization objects used to allow multiple threads to serialize access to shared data. Spinlocks are implemented as integer variables that are changed and tested with an atomic instruction. This simple approach yields fast operations; however, there are some side effects:

- ▶ Applications waiting for a lock actively consume CPU resources.
- ▶ There is no kernel protection for spinlocks. Any thread can change them using a simple assignment instruction.

Important: Spinlocks are used to control access to small critical sections which complete quickly. Incorrect use of spinlocks will have an adverse affect on system performance.

zSeries-specific features and spinlocks

On z/OS and earlier operating systems running on mainframe platforms, synchronization was often provided by means of COMPARE AND SWAP (CS) assembler instruction. CS can be used by programs sharing common storage areas in either multiprogramming or multiprocessing environments since the operation is atomic.

The kernel mechanism of spinlocks based on the CS instruction is shown in Example 15-5.

Example 15-5 Spinlocks and CS instruction (linux/include/asm-s390/spinlocks.h)

```
extern inline void spin_lock(spinlock_t *lp) {
    __asm__ __volatile("    bras    1,1f\n"
                       "0:   diag    0,0,68\n"
                       "1:   slr     0,0\n")
}
```

```

"    cs    0,1,%1\n"
"    jl    0b\n"
: "=m" (lp->lock)
: "0" (lp->lock) : "0", "1", "cc" );
}

```

The diag 68 (44 hex) instruction ends the current timeslice in a virtual machine. This instruction can be called only in a supervisor mode, so this implementation of spinlock is not useful for user mode.

Spinlock operations provided by pthread library

Unfortunately, pthread spinlocks are not described in man pages yet. You can find the POSIX specification at:

<http://www.opengroup.org/onlinepubs/007904975/basedefs/contents.html>

Bear in mind that not all of the described features are implemented in Linux.

Creating spinlocks in shared memory

In order to use spinlock, you should allocate shared memory for the spinlock variable (see Example 15-6).

Example 15-6 Creating spinlock in shared memory

```

#define _XOPEN_SOURCE 2000
#include<features.h>
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<pthread.h>

struct shmids ds;
pthread_spinlock_t *p;
int mid;

main(){
    printf("%d\n",mid=shmget(111, 100,IPC_CREAT | 0777));
    p=shmat(mid,0,0);
    pthread_spin_init (p, PTHREAD_PROCESS_SHARED);
}

```

Important: It is essential to properly define the macro `_XOPEN_SOURCE` (set to 2000) and to include the `features.h` header file *before* any other header file (it defines the `__USE_XOPEN2K` macro needed by the `thread_spin_` functions).

Using spinlocks

Example 15-7 shows how the spinlock functions are used to protect a critical section. This works for both threads and processes.

Example 15-7 Using spinlocks

```
#define _XOPEN_SOURCE 2000
#include<features.h>
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<pthread.h>
#include<bits/pthreadtypes.h>
#include<unistd.h>

pthread_spinlock_t *p;
int mid;

main(){
    mid=shmget(111, 100,IPC_CREAT | 0777);
    p=shmat(mid,0,0);
    while(1){
        printf("%5d: Waiting ...\n",getpid());
        pthread_spin_lock (p);
        printf("%5d: Entered ! \n",getpid());
        sleep(rand()%5 + 1);
        printf("%5d: Leaving ! \n",getpid());
        pthread_spin_unlock (p);
        sleep(rand()%5+1);
    }
}
```

Barrier functionality

A *barrier* is a synchronization object that allows multiple threads or processes to synchronize at a particular execution point. All threads or processes wait in the `pthread_barrier_wait` function until a given number of tasks (specified in the `pthread_barrier_init` function) reaches this point. This mechanism is also supported by the pthread library on Linux for zSeries. For details, see:

http://opengroup.org/onlinepubs/007904975/functions/pthread_barrier_wait.html

The pthread library - a problem

The version of the pthread we used (glibc-2.2.2 supplied with the SuSE SLES7 distribution) has a nasty bug. Look at the output from the **objdump** command:

```
objdump --disassemble /usr/lib/libpthread.a | less
```

Function `__pthread_spin_unlock` uses the register R1 instead of R2 when looking for a parameter (see 7.1.3, “Function calling convention” on page 93).

Example 15-8 Incorrect unlock

```
00000030 <__pthread_spin_unlock>:
 30: d7 03 10 00 10 00      xc      0(4,%r1),0(%r1)
 36: 07 f0                  br      %r0
 38: a7 28 00 00           lhi     %r2,0
 3c: 07 fe                  br      %r14
 3e: 07 07                  bcr     0,%r7
```

Recompiling glibc

To correct this problem, recompile glibc using the following steps:

1. Install the glibc.spm package.

2. Unpack the source tarballs:

```
cd /tmp
bunzip2 </usr/src/packages/SOURCES/glibc-2.2.2.tar.bz2 | tar xf -
cd glibc-2.2.2
bunzip2 </usr/src/packages/SOURCES/glibc-linuxthreads-2.2.2.tar.bz2 | \
tar xf-
```

3. Apply patches:

```
cd /tmp
tar zxvf usr/src/packages/SOURCES/glibc-2.2.2-s390.tar.gz
cd glibc-2.2.2
patch -p1 < ../glibc-2.2.2-s390.diff
```

4. Correct the linuxthreads/sysdeps/s390/pspinlock.c. The contents of the `__pthread_spin_unlock` function should look as follows:

```
int __pthread_spin_unlock (pthread_spinlock_t *lock){
    asm volatile("xc 0(4,%0),0(%0)\n"
                 "bcr 15,0"
                 : "=a"(lock) : "0" (lock) : "memory","cc");
    return 0;
}
```

5. Compile the library:

```
cd /tmp/glibc-2.2.2
./configure --enable-add-ons
make
```

Note: If you have problems with CVS commands in one of the **make** files, you can remove this line. We do not use it here.

Now you can copy or install the pthread library from the `glibc-2.2.2/linuxthreads` directory and use it in your programs.

Archived

Concurrency in embedded SQL

In this chapter we discuss embedded SQL programs in DB2 UDB on Linux for zSeries. We focus on multiple database connections within a single application. This topic is closely related to the pthreads synchronization described in the previous chapter.

16.1 Using embedded SQL in DB2 UDB applications

In this section we describe how DB2 UDB applications are built.

16.1.1 Components of a DB2 UDB application

An embedded DB2 SQL application consists of two parts:

- ▶ Application executable files
- ▶ A package stored in the database

A *package* is an object stored in the database that includes information needed to execute specific SQL statements in a single source file. A database application uses one package for every precompiled source file used to build the application. Each package is a separate entity, and has no relationship to any other packages used by the same or other applications. The package creation process is called *binding*.

Database applications use packages for improved performance and compactness. By precompiling an SQL statement, the statement is compiled into the package at bind time—not at a run time. Each statement is parsed, and a more efficiently interpreted operand string is stored in the package. At run time, the code generated by the precompiler calls run-time services in the database manager with any variable information required for input or output data. The information stored in the package is then executed.

Note: The advantages of precompilation apply only to static SQL statements. SQL statements that are executed dynamically (using PREPARE and EXECUTE or EXECUTE IMMEDIATE) are not precompiled. Therefore, they must go through the entire set of processing steps at run time.

16.1.2 Creating a package

In this section we describe the process of preparing the embedded SQL program shown on Figure 16-1.

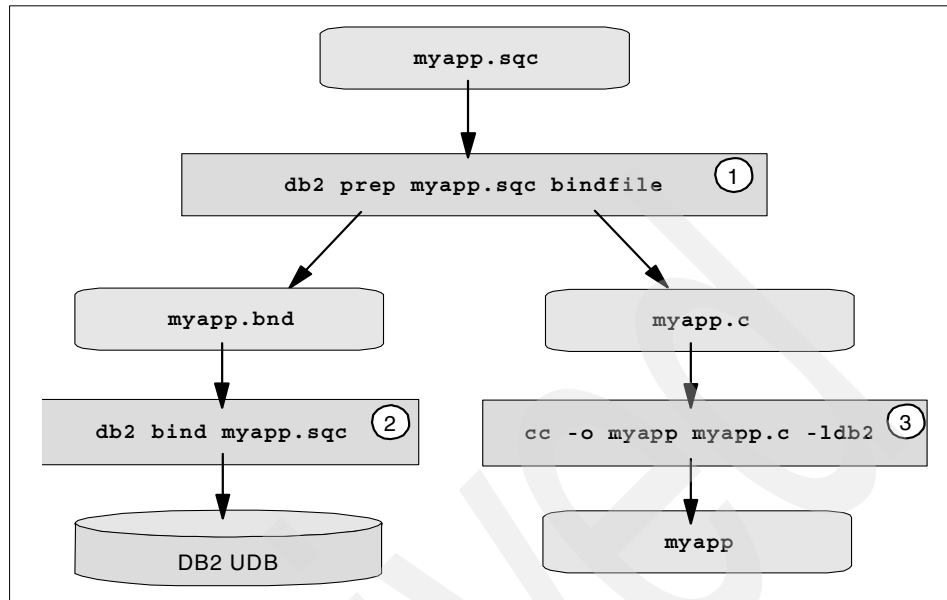


Figure 16-1 Creating a program with embedded SQL

Example 16-1 shows a very simple C program (myapp.sql) with static SQL statements.

Example 16-1 Simple C program with embedded SQL statements - myapp.sqc

```

01:  #include <stdio.h>
02:  #include <stdlib.h>
03:  #include <string.h>
04:  #include "utilemb.h"
05:
06:  EXEC SQL INCLUDE SQLCA;
07:
08:  int main(int argc, char *argv[])
09:  {
10:      EXEC SQL CONNECT TO SAMPLE USER db2inst1 USING ibmdb2;
11:      if(SQLCODE != 0){
12:          printf("Cannot connect to the database\n");
13:          exit(1);
14:      }
15:      EXEC SQL INSERT INTO T1 VALUES (1,2);
16:      EXEC SQL DISCONNECT SAMPLE;
17:  }
  
```

Line 6 includes some DB2 UDB-specific definitions and declares an instance of the variable:

```
struct sqlca sqlca;
```

This variable is used by the DB2 API which replaces EXEC SQL statements (lines 10,15-16) in the preparation stage.

For a comprehensive discussion of static SQL programming, refer to *IBM DB2 Universal Database Version 7 Application Development Guide*, SC09-2949.

As shown in Figure 16-1, the application creation consists of three steps:

1. The prep command converts embedded SQL statements into DB2 run-time API calls that a host compiler can process, and creates a bind file.

The bind file contains information on the SQL statements in the application program. Optionally, the precompiler can perform the bind step at precompile time (if you do not specify the `bindfile` option). We suggest you always create a separate bind file. Precompiling with deferred binding allows an application to access many databases because you can bind it against each one. This approach to DB2 application development is more flexible since it allows you to distribute only the binary executable along with the bind file to get your application running on another machine.

Tip: The bind file can be investigated with the `db2bfd` command. For example:

```
db2bfd -s myapp.bnd
```

lists all SQL statements include in the myapp.bnd file.

2. The BIND command creates a package in the database.
3. The final executable is created and linked with the DB2 libraries. Use the `-ldb2` option and ensure this library is in the linker search list.

16.1.3 Incorporating prep/bind into make

Since the DB2 program is prepared in at least three steps, it is convenient to use Makefile rules to go through the preparation and binding process. Example 16-2 quotes relevant lines from our Makefile.

Note that the `adrbook.h` file is generated automatically by the `db2dc1gn` utility based on the table declaration.

Example 16-2 Makefile rules to create bindfile, bind it and compile the C source

```
sqldb.o: sqldb.sqc itsodb.h sqldb.h adrbook.h utilemb.h
    db2 connect to $(DBNAME) user $(DBUSER) using $(DBPASS)
    db2 prep sqldb.sqc bindfile
    db2 bind sqldb.bnd
    db2 terminate
    $(CC) $(CCARG) -c sqldb.c
adrbook.h:
    db2dc1gn -d $(DBNAME) -u $(DBUSER) -p $(DBPASS) \
        -i -t adrbook -o adrbook_add.h -c -a replace
```

16.1.4 Embedded SQL files as libraries

In projects that consists of many executable files, the same database statements are often executed from different programs. In this case, you can create one single library that implements all the functions you need and hides the database implementation details. This library source file should be preprocessed and only this library should be bound against the database, regardless how many programs use it. The library may be static or dynamic.

Tip: We advise you to use libraries, even if the functions are relatively small. Libraries make your application more compact, and simplify subsequent maintenance or migration to a different database version.

If you want to use DB2 libraries in multithreaded applications, you must provide a means to synchronize threads' access to program data. However, do not attempt to serialize database access—databases are designed to accommodate multiple, concurrent clients.

In the next section we describe how the connection pooling is implemented within the address book library.

16.2 Multiple connections in embedded SQL programs

DB2 UDB provides an API to maintain more than one context within embedded SQL program. This allows a single threaded application to perform SQL operation beyond the current transaction, and for multiple threads to perform database functions simultaneously.

16.2.1 Connection context

Context refers to the current state of database connection.

Note: Each context can be seen as a single entry in the following db2 command:

```
db2 list application
```

Contexts are used to connect to, operate on, and disconnect from a database. Contexts may be shared by multiple threads. Figure 16-2 illustrates the logic flow for the `addUser` function, which adds a user in the sample application.

To provide an efficient solution, a constant pool of connections is maintained. When an application thread needs to perform a database operation, it acquires a free context. Example 16-3 shows the relevant data structures involved.

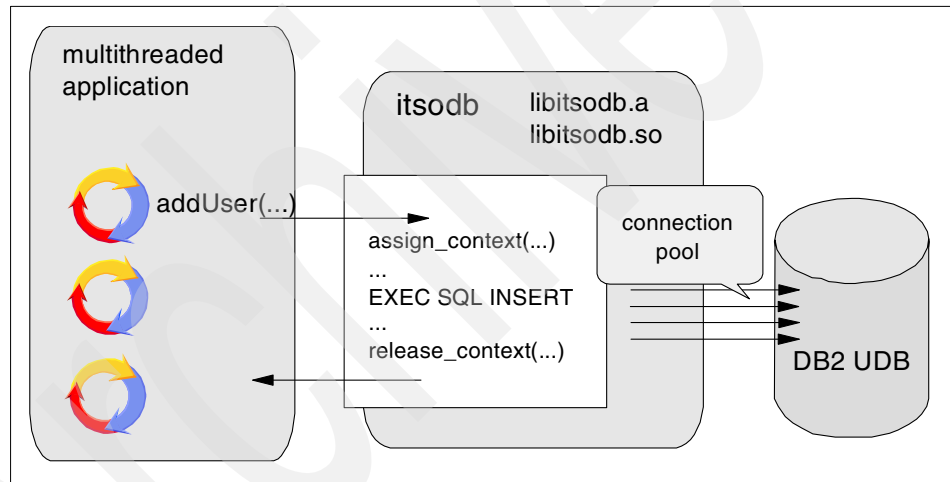


Figure 16-2 Working with contexts in multithreaded applications

16.2.2 Context operations

In this section, we describe how a pool of connections was implemented in the address book example.

Declarations

The `sql_ctx` array describes a pool of connections. Each context may be in one of the following states:

`CTX_NOTCONNECTED` The context is not initialized

CTX_FREE	The context is initialized and the database connection is established. The context can be used.
CTX_ADD	The context is used by the add_user function and becomes available as soon as the function ends.
CTX_SEARCH	The context is used by the search functionality and becomes unavailable until all records are retrieved.

Example 16-3 Maintaining a pool of DB2 connections - the definitions (sqldb.sqc)

```

#include<unistd.h>
#include<sql.h>
#include<pthread.h>
#include"itsodb.h"

EXEC SQL INCLUDE SQLCA;

/* .... */

enum { CTX_NOTCONNECTED, CTX_FREE, CTX_ADD, CTX_SEARCH };

struct sql_ctx{
    void *ptr;
    int state;
} sql_ctx[NUM_CTX];

int ctx_free = NUM_CTX;
int ctx_close_connections = FALSE;

pthread_mutex_t ctx_mutex;
pthread_cond_t  ctx_cond;

```

Initialization

Now we take a look at the initialization process shown in Example 16-4.

Important: The communication area has to be defined in each of the functions locally:

```
struct sqlca sqlca;
```

Example 16-4 Maintaining a pool of DB2 connections - the initialization(sqlldb.sqc)

```
/* *****
 * sqldb_open(char *conname)
 * - creates and opens NUM_CTX database contexts
 * - initializes mutual exclusion variables
 * ***** */
EXEC SQL BEGIN DECLARE SECTION;
    char dbname[16];
    char dbuser[16];
    char dbpass[16];
EXEC SQL END DECLARE SECTION;

static sqldb_parse(char *conname){

    /* parse argument - split it into three parts
       and move them into: dbname, dbuser, dbpass */
}

sqldb_open(char *conname){
    int i;
    struct sqlca sqlca;

    sqldb_parse(conname);
    pthread_mutex_init(&ctx_mutex, NULL);
    pthread_cond_init(&ctx_cond, NULL);
    sqleSetTypeCtx(SQL_CTX_MULTI_MANUAL);
    for (i=0; i < NUM_CTX; i ++) {
        sqleBeginCtx(&(sql_ctx[i].ptr),
                    SQL_CTX_BEGIN_ALL,
                    NULL,
                    &sqlca);
        ITSO_SQL_CHECK("sqleBeginCtx");
        if (SQLCODE != 0)
            return -1;
        EXEC SQL CONNECT TO :dbname USER :dbuser USING :dbpass;
        ITSO_SQL_CHECK("CONNECT");
        if (SQLCODE != 0)
            return -1;
        sqleDetachFromCtx(sql_ctx[i].ptr,
                        NULL,
                        &sqlca);
        ITSO_SQL_CHECK("sqleDetachFromCtx");
        if (SQLCODE != 0)
            return -1;
        sql_ctx[i].state=CTX_FREE;
    }
}
```

The DB2 API functions used in Example 16-4 are described in the following subsections.

sqlSetTypeCtx

The `sqlSetTypeCtx` function should be the first database call made inside an application. It sets the application context type. We use it enable contexts to be shared across threads (`SQL_CTX_ORIGINAL`).

```
int sqlSetTypeCtx(sqlint32 options);
```

Valid options values are:

<code>SQL_CTX_ORIGINAL</code>	Default setting. All threads will use the same context, and concurrent access will be blocked.
<code>SQL_CTX_MULTI_MANUAL</code>	All threads will use separate contexts, and it is up to the application to manage the context for each thread.

Note: Automatic COMMIT at process termination is disabled when using `SQL_CTX_MULTI_MANUAL` mode.

sqlBeginCtx

The `sqlBeginCtx` function creates an application context and optionally attaches the calling thread to that context.

```
int sqlBeginCtx(void **ppCtx,  
                sqlint32 options,  
                void *reserved,  
                struct sqlca *pstSqlca);
```

Parameters are:

<code>ppCtx</code>	Data area allocated for context information.				
<code>options</code>	There are two possible options: <table><tr><td><code>SQL_CTX_CREATE_ONLY</code></td><td>Context memory will be allocated, but there will be no attachment.</td></tr><tr><td><code>SQL_CTX_BEGIN_ALL</code></td><td>Context memory will be allocated, and then a thread is attached to the context by implicit <code>sqlAttachToCtx</code> call.</td></tr></table>	<code>SQL_CTX_CREATE_ONLY</code>	Context memory will be allocated, but there will be no attachment.	<code>SQL_CTX_BEGIN_ALL</code>	Context memory will be allocated, and then a thread is attached to the context by implicit <code>sqlAttachToCtx</code> call.
<code>SQL_CTX_CREATE_ONLY</code>	Context memory will be allocated, but there will be no attachment.				
<code>SQL_CTX_BEGIN_ALL</code>	Context memory will be allocated, and then a thread is attached to the context by implicit <code>sqlAttachToCtx</code> call.				
<code>reserved</code>	Reserved for future use. Must be set to NULL.				
<code>pstSqlca</code>	Pointer to the <code>sqlca</code> structure.				

Assigning and freeing context

For simplicity, we implement two basic database functions in our example:

- Add a new entry.

A record is inserted and commit automatically performed on completion.

- Search for and retrieve records

An initialization phase prepares a query returning a handle to the context.

A retrieval phase uses the handle to the next record. Records may be retrieved by the different threads. Threads are detached from the context but the transaction remains active until all records are read.

An operation performed by a library function may be an auto-committed transaction (in `add_user` for example) or may be part of a longer transaction (when retrieving records for example).

In the first case, we assign a context and use that context for subsequent SQL operations. Upon completing SQL operations, the transaction is committed by the database engine (using auto-commit). The context is then returned to the pool where it becomes available for use by another thread.

In the second case, we begin a transaction (by opening a cursor, for example) on a threaded request. The context is then detached using the `sqlDetachFromCtx` function (causing the context to be marked as `CTX_SEARCH`) and the threaded request completes. The transaction remains uncommitted however. The context id is returned in order to obtain the desired context for subsequent SQL operations (later threaded requests will first obtain the correct context for SQL operations using the context id). When all threaded requests complete, the transaction will be explicitly committed. At that time, the context is marked `CTX_FREE` and returned to the pool (thus making it available to process another transaction).

We implemented four functions in order to provide this functionality:

<code>assign_ctx</code>	Attach to a free context.
<code>release_ctx</code>	Commit and detach from the context and mark it as a free one.
<code>join_ctx</code>	Attach to the context already assigned to an opened search transaction.
<code>join_ctx</code>	Detach from the context but do not commit the transaction, and leave the context state set to <code>CTX_SEARCH</code> .

In the next sections we describe only `assign_ctx` (Example 16-5) and `release_ctx` (Example 16-6). The code regarding DB2 connections does not change in `join_ctx` and `join_ctx` significantly.

sqlAttachToCtx

Use of the `sqlAttachToCtx` function is straightforward.

Example 16-5 Maintaining a pool of DB2 connections - assign_ctx(sqlcldb.sqc)

```
int assign_ctx(int action) {
    int i;
    struct sqlca sqlca;

    /* wait for a free context <- pthread related section was removed !!!*/
    for (i=0; i < NUM_CTX; i++)
        if(sql_ctx[i].state == CTX_FREE){
            sqlAttachToCtx(sql_ctx[i].ptr,
                           NULL,
                           &sqlca);
            ITSO_SQL_CHECK("sqlAttachToCtx");
            if (SQLCODE != 0)
                return -1;
            sql_ctx[i].state = action;
            ctx_free--;
            return i;
        }
    return -1;
}
```

The `sqlAttachToCtx` function allows a thread to join a context. If more than one thread is attached to a given context, access is serialized for these threads, and they share a commit scope.

```
int sqlAttachToCtx(void *pCtx,
                  void *reserved,
                  struct sqlca *pstSqlca);
```

where parameters have the following meanings:

<code>pCtx</code>	Valid context previously allocated by <code>sqlBeginCtx</code> .
<code>reserved</code>	Reserved for future use (must be set to NULL).
<code>pstSqlca</code>	Pointer to the <code>sqlca</code> structure.

sqlDetachFromCtx

The `sqlDetachFromCtx` function detaches the context being used by the current thread. Example 16-6 shows how the `sqlDetachFromCtx` function was used to implement `leave_ctx`.

Example 16-6 Maintaining a pool of DB2 connections - leave_ctx(sqldb.sqc)

```
int release_ctx(int ctx){
    struct sqlca sqlca;

    sqlDetachFromCtx(sql_ctx[ctx].ptr,
                     NULL,
                     &sqlca);
    /* code maintaining concurrent access was removed */
    ITSO_SQL_CHECK("sqlDetachFromCtx");
    if(SQLCODE != 0)
        return -1;
    sql_ctx[ctx].state = CTX_FREE;
    ctx_free++;
    if (ctx_close_connections == TRUE)
        sqldb_close_ctx(ctx);
    return 0;
}
```

```
int sqlDetachFromCtx(void *pCtx,
                    void *reserved,
                    struct sqlca *pstSqlca);
```

Parameters of `sqlDetachFromCtx` have the following meanings:

<code>pCtx</code>	Valid context previously allocated by <code>sqlBeginCtx</code> .
<code>reserved</code>	Reserved for future use. Must be set to <code>NULL</code> .
<code>pstSqlca</code>	Pointer to the <code>sqlca</code> structure.

Finalization

When the connection is no longer needed it is closed with the SQL command:

```
EXEC SQL DISCONNECT database;
```

Finally, the context memory may be released. This is shown in Example 16-7.

```
int sqlldb_close_ctx(int ctx){
    struct sqlca sqlca;
    sqleAttachToCtx(sql_ctx[ctx].ptr,
                    NULL,
                    &sqlca);
    ITSO_SQL_CHECK("sqleAttachToCtx");
    EXEC SQL DISCONNECT :dbname;
    ITSO_SQL_CHECK("DISCONNECT");
    sqleEndCtx(&(sql_ctx[ctx].ptr),
               SQL_CTX_END_ALL,
               NULL,
               &sqlca);
    ITSO_SQL_CHECK("sqleEndCtx");
    ctx_free--;
}
```

sqleEndCtx

Call this function to free all memory associated with a given context. The context must not be used by another thread.

```
int sqleEndCtx(void **ppCtx,
               sqlint32 options,
               void *reserved,
               struct sqlca *pstSqlca);
```

The parameters have the following meanings:

ppCtx	Pointer to the context pointer previously set by <code>sqleBeginCtx</code> .
options	Valid values are: SQL_CTX_FREE_ONLY Context memory will be freed only if a prior detach has been done. SQL_CTX_END_ALL Call <code>sqleDetachFromCtx</code> , if necessary.
reserved	Reserved for future use. Must be set to NULL.
pstSqlca	Pointer to the <code>sqlca</code> structure.

In our example, we do not release the connection when it is used by the other thread. The `ctx_close_connection` variable is set instead, and the context will be detached on the subsequent `release_ctx` call.

16.2.3 Client-server considerations

Maintaining a connection pool may be used in client-server applications when you do not want to, or for some reasons cannot use DB2 Runtime clients.

An approach such as that shown on Figure 16-3 might be used when:

- ▶ You want to hide all knowledge of database operations from the clients.
- ▶ The DB2 runtime files cannot be installed on the client machines.
- ▶ The driver does some additional work apart from the database which cannot be implemented as stored procedures.
- ▶ The client applications do not send messages frequently and you do not want to maintain idle database connections (or keep your database server busy with connect and disconnect operations).

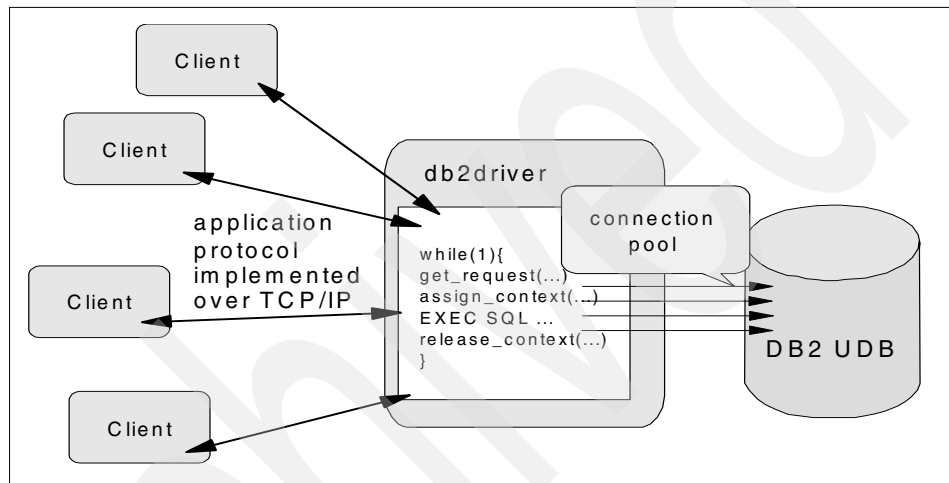


Figure 16-3 Connection pooling with client-server scenario.

However, you should have strong reasons to design the application in this way. Remember that you are building your *own* DB2 driver.

Packaging applications for deployment

This chapter describes how to configure and package our sample application for deployment. Topics covered are:

- ▶ The application source structure and the prerequisite software needed to build the application.
- ▶ How to build and package the C portion in RPM (Redhat Package Manager) format for deployment.
- ▶ How to build and package the Java portion in WAR (Web Application aRchive) format for deployment.

17.1 Creating a project

The first step in creating a project is to determine the directory structure for the application source files. Using the sample applications, we illustrate a structure that conforms to the Java Servlet specification for Web application deployment. The specification can be found at:

<http://java.sun.com/products/servlet/download.html>

Some important tools we use here are:

- ▶ CVS - to maintain revision control (see 3.7, “Creating a project” on page 41)
- ▶ Ant - to build Java components (see 6.2.2, “Using Ant” on page 80)
- ▶ make - to build C components (see 1.4.1, “GNU make” on page 14)

17.1.1 Example source structure

The source code comprising the sample project is arranged as follows:

sg246807	Project root directory
lib	Directory for external libraries
src	Source code directory
web	Web application directory

Project root directory

In the project root directory, we added the following files used by Ant to create the project:

build.xml	The Ant build descriptor file
build.properties	Additional properties file included by build.xml

See 6.2.2, “Using Ant” on page 80 for an overview of using Ant.

Library directory

The `lib` directory will contain any external libraries (`.jar` files) referenced by the application. For example, the `struts.jar` library file is placed in this directory, along with any taglibs used by the application (see 6.4.1, “Installing taglibs” on page 82). Ant will place the contents of this directory into the `WEB-INF/lib` directory of the Web application archive (`.war`) file.

Source directory

The application source consists of both Java and C/C++ code placed in an appropriate subdirectory.

Java source directory

The Java directory consists of the source files that comprise the Struts example application. (See Chapter 13, “Using the Struts framework” on page 167 for complete details about the files referenced here). The following additional application property files exist in this directory:

<code>AppResources.properties</code>	The application resource descriptor
<code>log4j.properties</code>	The Log4j configuration descriptor
<code>sg246807.properties</code>	The application persistence descriptor

Java class files are placed in subdirectories based on their function:

<code>action</code>	Struts Action sub-classes
<code>bo</code>	The application business logic
<code>data</code>	Application persistence classes
<code>form</code>	Struts ActionForm sub-classes

C source directory

The `c` directory contains the `JniAddressbook.c` implemented for JNI database access. Additionally, the shared library and Qt application implementation files are found in the following subdirectories:

<code>addressbook-1.2</code>	The Qt front-end to the Address book application
<code>db2driver</code>	The threaded embedded SQL example
<code>itsodb-1.2</code>	The example shared library for accessing DB2
<code>rpm-specs</code>	RPM specification files to build rpm packages

Web application directory

The Web directory contains files used in deployment on an Java application server: JSP pages and the `WEB-INF` subdirectory.

WEB-INF directory

The `WEB-INF` directory contains the following application deployment files:

<code>struts-config.xml</code>	The Struts configuration descriptor
<code>web.xml</code>	The application deployment descriptor

17.1.2 Adding prerequisite libraries to the project

The sample project relies on some external libraries. These include:

- ▶ Struts framework
- ▶ Log4j Jakarta taglib
- ▶ JDBC 2.0 compliant driver (we used the driver supplied with DB2)

In addition, you will need to install Ant and a Java application server (Tomcat, for instance). Follow these steps to install the prerequisites:

1. Install Ant (see 6.2.1, “Installing Ant” on page 80).
2. Install the Tomcat server (see 6.1.2, “Installing Tomcat” on page 76).
3. Install Struts to the project (see 6.5.2, “Installing Struts” on page 87).
 - a. Copy the `struts.jar` library to the `sg246807/lib` directory.
 - b. Copy the `struts-*.tld` taglib descriptor files to the `sg246807/web/WEB-INF` directory.
4. Install Log4j to the project (see 6.3.1, “Installing Log4j” on page 81).
 - a. Copy the `log4j-1.2rc1.jar` library to the `sg246807/lib` directory.

17.1.3 Prepare the database

To use a relational database, the application server requires a JDBC 2.0 compliant driver to be added to its CLASSPATH. We used the driver supplied with DB2.

1. Install DB2 (see “Installing DB2” on page 268).
2. Copy the DB2 JDBC driver (\$DB2DIR/java12/db2java.zip) to the Tomcat common/lib directory. A symbolic link works:

```
ln -s $DB2DIR/java12/db2java.zip /
    $CATALINA_HOME/common/lib/db2java.jar
```

Important: The driver *must* be named db2java.jar. Tomcat only adds .jar files to its CLASSPATH.

3. Create a database named SAMPLE.
4. Add a table named ADRBOOK to SAMPLE (see Example 17-1). A sample script (sg246807/src/c/itsodb-1.2/crt_tables.db2) can be used to create this table.

Example 17-1 ADRBOOK table (sq246807/src/c/itsodb-1.2/crt tables.db2)

```
CREATE TABLE ADRBOOK(
```

```
FNAME    VARCHAR(16) NOT NULL,  
LNAME    VARCHAR(32) NOT NULL,  
PHONE    VARCHAR(16),  
EMAIL    VARCHAR(32)  
);
```

17.1.4 Customize the application

To complete the setup, you need to customize some property files:

- ▶ `sg246807/src/java/log4j.properties`
Provide e-mail configuration parameters for logging (see Example 13-7 on page 177).
- ▶ `sg246807/src/java/sg246807.properties`
Provide database configuration parameters (see Example 13-14 on page 186).

17.2 Creating RPM packages

The RPM-HOW-TO available at:

<http://www.tldp.org/HOWTO/RPM-HOWTO/>

contains a detailed description of the package creation process. In the following section, we describe how we built the package for the address book library and the standalone application.

Example 17-2 RPM specification file (itsodb.rpmspec)

```
Summary: A simple address book library  
Name: itsodb  
Version: 1.2  
Release: 1  
Copyright: IBM  
Group: Development/Libraries  
Source: itsodb-1.2.1.tgz
```

```
%description
```

```
The address book example library as described in  
"Linux for zSeries: Application Development"
```

```
%prep  
%setup -q  
make crttab
```

```

%build
make

%install
make install-devel PREFIX=/usr/local
%post
ldconfig
%postun
ldconfig

%clean
make uninstall PREFIX=/usr/local

%files
/usr/local/itsodb
/usr/local/lib/libitso*

```

17.2.1 Before you begin

Before creating the RPM specification file, you should decide on:

- ▶ A package name
- ▶ The package version
- ▶ The package contents
- ▶ The destination directories for your files. We used the following:

<code>/usr/local/itsodb</code>	Installation base
<code>/usr/local/lib</code>	Links to the libraries
<code>/usr/local/itsodb/include</code>	Header files. This directory should be added to the search list when compiling applications using this library (gcc option <code>-I</code>)

17.2.2 Preparing the source archive

The source directory should be named `packagename-version` according to the RPM specification file. In our case the name is `itsodb-1.2`.

Prepare the source archive:

1. Remove all unnecessary files from this directory:

```

cd itsodb-1.2
make clean

```

2. Create the `tar` archive `packagename-version.release`:

```
cd ..  
tar zcvf itsodb-1.2.1.tgz
```

3. Copy the archive to the `/usr/src/packages/SOURCE` directory:

```
cp itsodb-1.2.1.tgz /usr/src/packages/SOURCES/
```

17.2.3 Preparing package specification

The RPM specification shown in Example 17-2 present the following sections:

- ▶ General information
 - Package name and version
 - Description of the contents
 - Package category (group)

Note: This section may contain the build directory used when building a binary archive. We left it out here for simplicity, but we advise you to use this option.

- ▶ Prepare section: Invoked before building the binary package.
 - `setup -q` - unpacks the archive and changes the current directory to it.
 - `crttable` is the make rule for `itsodb` which creates tables in the database.
- ▶ Build: We invoke the make here.
- ▶ Install: We rely on the make rule `install-devel` which installs the libraries and header files.
 - `post` rules are to be invoked after the installation of a binary package.
 - `postun` rules are to be invoked after the package is removed from the system.
- ▶ Clean: Remove all files created during this process. Note that if you do not specify a build root, the RPM uses your base files system (for example, `/usr/lib` directories).
- ▶ Files: This section describes all the files that are included in the binary archive. It is not possible to link the make `install` rule with this process.

17.2.4 Building the package

Issue the following command to produce the packages:

```
rpm -ba itsodb.rpmspec
```

This creates two files:

```
/usr/src/packages/SRPMS/itsodb-1.2-1.src.rpm
/usr/src/packages/RPMS/s390/itsodb-1.2-1.s390.rpm
```

You can also build the packages for different architectures provided that the cross-compiler and the libraries were installed. Use the `--target` option in this case.

17.2.5 Installing packages

Check the archive contents:

```
rpm -qilp itsodb-1.2-1.s390.rpm
```

The package is produced but not installed. Use the standard `rpm` options to install it:

```
rpm -U itsodb-1.2.1.s390.rpm
```

This installs the library in `/usr/local/itsodb/` and creates the proper links from `/usr/local/lib`. Some other useful files are copied, too.

<code>crt_tables.db2</code>	DB2 UDB script to prepare the table
<code>binder.sh</code>	Shell scripts to bind the library package to a database
<code>include/*.h</code>	Include files for C programs

17.3 Creating WAR packages

A Web application archive (WAR) is a compressed package of a directory with the following structure:

<code>web-app</code>	The application root directory containing HTML, JSP, style sheets, and other files to be served on client request.
<code>WEB-INF</code>	The Web resources directory.
<code>classes</code>	Java class file directory containing application classes and <code>.jar</code> files.
<code>lib</code>	Java library file directory containing external libraries required by the application.

Note: The `WEB-INF` directory is inaccessible to client HTTP requests.

WAR packages are the standard mechanism for deploying Java servlets.

17.3.1 Building a WAR package using Ant

Ant provides a built-in task to create WAR packages—the war task. Example 17-3 shows the process in detail; notes following the example explain some significant parts.

Example 17-3 Building a WAR package (build.xml)

```
<?xml version="1.0"?>
<project name="sg246807" default="dist" basedir=".">
  <property name="src.base" value="src"/>
  <property name="src.java.dir" value="${src.base}/java"/>
  <property name="src.c.dir" value="${src.base}/c"/>
  <property name="web.dir" value="web"/>
  <property name="build.dir" value="build"/>
  <property name="dist.dir" value="dist"/>
  <property name="build.classes" value="${build.dir}/WEB-INF/classes"/>
  <property name="build.lib" value="${build.dir}/WEB-INF/lib"/>
  <property name="lib.dir" value="lib"/>
  <property file="${web.dir}/release.properties"/>
  <property file="build.properties"/>
  <property file="${user.home}/build.properties"/>

  <target name="init">
    <tstamp/>
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes}"/>
    <mkdir dir="${dist.dir}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${src.java.dir}" destdir="${build.classes}"
      classpathref="classpath"/>
  </target>

  <target name="build" depends="compile">
    <copy todir="${build.classes}">
      <fileset dir="${src.java.dir}">
        <include name="**/*.properties"/>
      </fileset>
    </copy>
    <copy todir="${build.lib}">
      <fileset dir="${lib.dir}">
        <include name="*.jar"/>
      </fileset>
    </copy>
    <copy todir="${build.dir}">
      <fileset dir="${web.dir}">
        <include name="**"/>
      </fileset>
    </copy>
  </target>
</project>
```

```

        </fileset>
    </copy>
</target>

<target name="dist" depends="build"
        description="Create WAR-File for JSP 1.1/Servlet Engine 2.2">
    <war warfile="${dist.dir}/sg246807-${release.name}.war"
        webxml="${build.dir}/WEB-INF/web.xml">
        <fileset dir="${build.dir}"/>
    </war>
</target>
</project>

```

6

1. The `<project>` stanza defines the Web application name (sg246807) and defines a default target (dist).
2. Various `<property>` definitions are provided.
3. The `init` target creates directories:

build	Web resource files (JSP, HTML, etc.)
build/classes	Application classes
dist	Application WAR package
4. The `compile` target compiles Java source files in the `src/java` directory and places the resultant class files in the `build/classes` directory. It depends on the `init` target.
5. The `build` target depends on `compile`, and creates a directory structure in the `build` directory which matches the deployment structure.
6. The `dist` target, which depends on `build`, creates a WAR package in the `dist` directory.

Checking library dependencies

To compile and run the example application, you need external Java libraries and the installed RPM `itsodb-1.2-1.s390.rpm`. The RPM is for the C part of the application. Example 17-4 shows the portion of the build file that checks these dependencies.

Example 17-4 Checking library dependencies using Ant (build.xml)

```

<target name="-show_error" unless="required_libs_available">
    <echo message="** ERROR **"/>
    <echo message="required libraries not found"/>
    <echo message="please read lib/README.TXT"/>
    <echo message="**"/>
    <fail message="Required Java libraries are missing."/>
</target>

```

1


```

<!-- Check library dependencies for native support -->
<target name="-rpm_cond" if="native.support">
    <exec executable="rpm" failonerror="true">
        <arg line="-V --nodeps itsodb-1.2-1"/>
    </exec>
</target>

<!-- Check library dependencies for Java code. -->
<target name="-lib_cond">
    <condition property="required_libs_available">
        <and>
            <available classname="org.apache.log4j.Logger"
                classpathref="classpath"/>
            <available classname="org.apache.struts.action.ActionMapping"
                classpathref="classpath"/>
        </and>
    </condition>
</target>

<target name="check_libs" depends="-lib_cond, -rpm_cond, -show_error"
    description="Compile Java and C code."/>
<target name="compile_java" depends="init, check_libs"
    if="required_libs_available" description="CompileJava">
    <javac srcdir="${src.java.dir}" destdir="${build.classes}"
        classpathref="classpath"/>
</target>

```

1. Target `-show-error` prints an error message to the console if the property `required_libs_available` does not evaluate to true.
2. Target `-rpm_cond` is executed if property `native.support` is true. It checks for existence of the `itsodb-1.2.1` RPM package using an `<exec>` rule to execute the `rpm` command. Ant terminates if the command fails to find the package.
3. Target `-lib_cond` checks for the existence of required library jar files in the CLASSPATH.
4. Target `-check_libs` simply ensures the previous three rules are executed in the correct order.
5. Target `-compile_java` compiles the Java source files. Its dependency on the `-check_libs` initiates the checks before compilation.

17.3.2 Deploying WAR packages

Use these steps to deploy the application:

1. Copy the WAR package to the Tomcat Web applications directory (naming it `sg246807.jar`):

```
cp sg246807/dist/sg246807-release.war /var/tomcat4/webapps/sg246807.jar
```

2. Restart the Tomcat server.

The application can then be accessed using:

```
http://localhost:8180/sg246807
```

Configuring a server for JDBC

To access a relational database in the application using JDBC, the JDBC interface needs to be configured in the application server. For DB2, the steps are:

1. Copy the DB2 JDBC (`db2java.zip`) driver to `$CATALINA_HOME/common/lib` directory:

```
cp instance/sql1lib/java12/db2java.zip /var/tomcat4/common/lib/db2java.jar
```

Important: Use a `.jar` file extension when installing the driver. Tomcat will not add `.zip` files to its CLASSPATH

Note: Use a symbolic link if you prefer:

```
ln -s instance/sql1lib/java12/db2java.zip \
/var/tomcat4/common/lib/db2java.jar
```

Note: The driver belongs in the server CLASSPATH - not the application CLASSPATH. Do *not* put the driver in the application `WEB-INF/lib` directory.

2. To use connection pooling, copy the Struts `jdbc2_0-stdext.jar` library to the `/var/tomcat4/common/lib` directory.

Configuring a server for JNI database access

To access a relational database in the application using the JNI interface, the server needs some configuration:

1. Build and install the `itsodb` RPM package.
2. Copy the `libJniAdressbook.so` shared library to the server `/var/tomcat4/common/lib` directory.

3. Edit the Tomcat startup configuration file (/etc/tomcat4/conf/tomcat4.conf) to make the DB2 libraries available to the loader. Add the following lines:

```
source db2instance/sql1lib/db2profile
export LD_LIBRARY_PATH=/var/tomcat4/common/lib
```

Configure the connection type

Finally, you will need to configure the application to use the appropriate connection type (see Section 13.9.1, “Connection configuration” on page 186).

17.3.3 Deployment on WebSphere Application Server 4.0

The sample application can be deployed using WebSphere Application Server 4.0. To obtain a trial version, go to:

<http://www14.software.ibm.com/webapp/download/search.jsp?go=y&rs=wasael>

To build the sample application using WebSphere Application Server, you need to specify the correct `servlet.jar` file to Ant. This property is defined in the `build.properties` file (see , “Project root directory” on page 252) and should be set to `/opt/WebSphere/AppServer/lib/j2ee.jar`. In addition, to use the WebSphere Application Server JDBC 2.0 connection pooling mechanism, specify the `jdb20_opt.jar` property as `/opt/WebSphere/AppServer/lib/j2ee.jar`.

Connection pooling using Java Naming Directory Interface

WebSphere Application Server supports the Java Naming Directory Interface (JNDI) to access a datasource. To use JNDI connection pooling with WebSphere Application Server 4.0:

1. Start the WebSphere Application Server admin client using the command:

```
$WAS_HOME/bin/admc1ient.sh
```

2. Create a data source by navigating **Console -> New -> Data Source**. Define the data source as:

```
NAME = sg246807
JNDI name = jdbc/sg246807
user = db2inst1
password = ibmdb2
```

3. Specify the `db.driverclass.dev` property in the `sg246807.properties` file (see Section 13.9.1, “Connection configuration” on page 186) as:

```
db.driverclass.dev=COM.ibm.db2.jdbc.DB2CConnectionPoolDataSource
```

For details on connection pooling, see:

<http://www-3.ibm.com/software/webservers/studio/appserver40pooling.html>

Deploying the application

To deploy the application:

1. Start the WebSphere Application Server admin client using the command:

```
$WAS_HOME/bin/admclient.sh
```

2. Install the application WAR file:

- a. Navigate **Wizard -> Install Enterprise Application Server**

- b. Install the module using the default values and:

```
Path = $SG24_HOME/dist/sg246807-release.war
```

```
Application name = sg246807
```

```
Context root for module = /sg246807
```

- c. Select resource sg246807 for the resource reference.



Part 4

Appendixes

Archived

DB2 for Linux on zSeries

This appendix provides instructions to install and set up DB2 on Linux for zSeries. This appendix refers to DB2 UDB v7.1.

Installing DB2

You must be logged on as root user in order to perform the installation.

Before you begin

IBM offers a 90-day trial copy of DB2 UDB EE (Enterprise Edition) for Linux for S/390 and zSeries on the Internet, available at:

<http://www6.software.ibm.com/dl/db2udbd1/db2udbd1-p/>

Download DB2 UDB EE on Linux for S/390 and zSeries V7.1 for Linux. You can also select the required language for the software product.

About 150 MB is needed to install DB2 UDB EE. Another 100 MB is necessary if the DB2 HTML Manuals are installed. The target directory for the db2setup program is /usr. In this directory an entry named /usr/IBMdb2/V7.1 is created. Most of the installed files are located there; however, db2setup adds some data in /var and /etc directories as well.

The installation files should be accessible from the Linux image on which to install. You can:

- ▶ Copy the installation archive to the target machine and unpack it.
- ▶ Use the VM shared disk.
- ▶ Use a PC with an NFS-exported directory.

We assume the installation files for db2 are located in the /db2install directory.

The contents of the installation directory look like the following:

```
.   NetData  db2_deinstall  db2setup  doc.cmn
..  db2      db2_install   doc       readme.txt
```

Prerequisites

Only one non-standard package is needed—the Korn Shell. We used the Public Domain Korn Shell implementation, which comes with most distributions; it can also be download from:

<http://rpmfind.org>

The package is named pdksh and is located in cd1/suse/ap2. Install it using YaST or the rpm command:

```
rpm -U pdksh.rpm
```


Installation procedure

Following is the list of the installation steps to be performed:

1. Start the installation program:

`./db2setup`

Note: The setup program may abend because it needs the `libstdc++-libc6.1-2.so.3` library. Follow the procedure described in “Standard C and C++ libraries” on page 18.

Tip: If the installation is interrupted, the `/tmp/.db2inst.1ck` lock file has to be removed.

2. The main screen of the installation program is shown in Figure A-1.

```
+----- Install DB2 V7 -----+
|
| Select the products you are licensed to install. Your Proof of
| Entitlement and License Information booklet identify the products for
| which you are licensed.
|
| To see the preselected components or customize the selection, select
| Customize for the product.
| [ ] DB2 Run-Time Client           : Customize... :
| [ ] DB2 UDB Enterprise Edition    : Customize... :
| [ ] DB2 Connect Enterprise Edition : Customize... :
| [ ] DB2 Application Development Client : Customize... :
|
| To choose a language for the following components, select Customize for
| the product.
|   DB2 Product Messages           [ Customize... ]
|   DB2 Product Library            [ Customize... ]
|
| [ OK ]                           [ Cancel ]                       [ Help ]
|
+-----+

```

Figure A-1 DB2 initial installation screen

3. Select the product you would like to use:
 - If you plan to install a standalone server, select **DB2 UDB Enterprise Edition**.

- If you want to connect to remote databases (either DB2 UDB for distributed platform, or DB2 on a host system like z/OS or iSeries), select **DB2 Connect**.
 - If you want to develop a DB2 application (regardless of the database connection type), also select **DB2 Application Development Client**.
4. Next, choose **OK** and proceed to the next screen, shown in Figure A-2.

```

+----- Create DB2 Services -----+
|
| Select the items you want to create, and select OK when finished.
|
| A DB2 Instance is an environment where you store data and run
| applications. An instance can contain multiple databases.
|
| ( ) Create a DB2 Instance.                                : Customize... :
| (*) Do not create a DB2 Instance.
|
| An Administration Server provides services to support client tools that
| automate the configuration of connections to DB2 databases.
|
| ( ) Create the Administration Server.                      : Customize... :
| (*) Do not create the Administration Server.
|
|
| [ OK ] [ Cancel ] [ Help ]
|
+-----+

```

Figure A-2 DB2 instance creation screen

5. It is often convenient to create a database instance during installation. An *instance* is an isolated environment where you can create and manage databases.

Tip: You can also create an instance later with the **db2icrt** command located in the `/usr/IBMDB2/V7.1/instance` directory. Refer to the *DB2 UDB Database Administration Guide* for more details.

An instance is always associated with a user account. The installation program will create such an account and configure a TCP/IP connection to your instance (port 50000). The details can be changed with the **Customize** option shown in Figure A-3.

```

+----- Create DB2 Services -----+
| Select the items you want to create, and select OK when finished. |
+--- DB2 Instance ---+
|
| Authentication:
|   Enter User ID, Group ID, Home Directory and Password that will be
|   used for the DB2 Instance.
|
|   User Name      [db2inst1]
|   User ID       :      :      [*] Use default UID
|   Group Name    [db2iadm1]
|   Group ID      :      :      [*] Use default GID
|   Home Directory [/home/db2inst1 ]
|   Password      [      ]
|   Verify Password [      ]
|
| Select Properties to view or change more options. [ Properties... ]
|
| Select Default to restore all default settings. [ Default ]
|
| [ OK ] [ Cancel ] [ Help ]
+-----+

```

Figure A-3 DB2 services installation screen

6. You need Administration Server when you want to manage your databases remotely. In our configuration we did not create an Administration Server instance.
7. On the next few screens, do *not* set the following unless you plan to use them:
 - DB2 Warehouse Control Database
 - DB2 Distributed Join for DB2 Data Sources
8. Select **OK** on the screen shown. The following warning may be ignored because we do not want to set up an Administration Server:

DBI1755W The Administration Server is not created.

Select **OK** to proceed to the next screen.
9. Select **Continue -> OK** in order to start the installation.

Note: The installation log is stored in the `/tmp/db2setup.log` file, and can be viewed online with following command:

```
tail -f /tmp/db2setup.log
```

In case of an error, this file contains a lot of diagnostic information you can refer to, as well as indications of which commands should be invoked to complete the process.

Configuring DB2

In our scenario, we used the single instance `db2inst1` created during installation. We created this instance on every Linux image.

Before you perform customizing, log on as the instance owner. The default is user `db2inst1` and the password is `ibmdb2`.

Creating a local database

We used the sample database provided with the DB2 UDB. To create the sample database, issue:

```
db2samp
```

You can connect to the sample database by using the DB2 command line interface. Enter the following:

```
$db2 connect to sample
Database Connection Information
Database server =DB2/LINUX 7.1.0
SQL authorization ID =DB2INST1
Local database alias =SAMPLE
```

Issue a sample query to verify the installation:

```
db2 "select * from staff"
```

Important: If you want to access the instance from a different user account, set up the environment by reading the `db2profile` (use the dot and space at the beginning):

```
. /home/db2inst1/sqllib/db2profile
```

Accessing remote databases

Before you access the data located on the remote host, you should catalog the node and the database.

UNIX or Windows database

If the remote side is a UNIX (including Linux for zSeries as shown in this example) or Windows environment, prepare the script as follows:

```
catalog tcpip node nodename
      remote server_ip_address
      server server_port_number
      ostype linux;
catalog db remotedbname as mydbname at node nodename
      authentication server;
```

And run in the db2 shell:

```
db2 -t -f scriptname
```

Tip: You can check the port the remote instance is listening on. Working as an ADM on the remote site, issue:

```
db2 get dbm cfg | grep SVC
```

The SVCENAME attribute contains the port number or the service name form /etc/services.

DB2 UDB for z/OS or s/390 databases

If the remote side is a z/OS or s/390 environment, execute the following script:

```
catalog tcpip node nodename remote server_ip_address
      server server_port_number ostype mvs;

catalog dcs db remotedbname ;

catalog db remotedbname as mydbname at node mfnode
      authentication dcs;
```

You can find *remotedbname* and the port number in DB2 log in SDSF. Look for the DSNL004I message and use the location and port fields, respectively; refer to Example A-1 on page 274.

Example: A-1 Obtaining the location and the port number

```
Display  Filter  View  Print  Options  Help
-----
SDSF OUTPUT DISPLAY DSNAMSTR STC00056  DSID   2 LINE 40    COLUMNS 02- 81
COMMAND INPUT ==>                                SCROLL ==> CSR
07.45.25 STC00056  DSNL519I @ DSNLILNR TCP/IP SERVICES AVAILABLE
                                FOR DOMAIN os01.appdev.net AND PORT 5021
07.45.25 STC00056  DSNL519I @ DSNLIRSY TCP/IP SERVICES AVAILABLE
                                FOR DOMAIN os01.appdev.net AND PORT 5022
07.45.25 STC00056  DSNL004I @ DDF START COMPLETE
                                LOCATION OS01DB
                                LU      ITSOSITE.OS01LU
                                GENERICLU -NONE
                                DOMAIN  os01.appdev.net1
                                TCPPORT 5021
                                RESPORT 5022
07.45.25 STC00056  DSN9022I @ DSNYASCP 'START DB2' NORMAL COMPLETION
00.00.00 STC00056 ---- FRIDAY,    24 MAY 2002 ----
```

Porting applications to Linux on zSeries

This appendix describes efforts to port applications to Linux on zSeries, and some application developer's tools available for Linux on zSeries. It is extracted from a number of white papers and Internet postings.

Linux for S/390 and zSeries porting - hints and tips

Note: This section was contributed by the S/390 Porting & Feasibility department, IBM Poughkeepsie, N.Y.

This section summarizes a variety of issues that our Linux Porting team has had to consider when doing ports of applications and tools to S/390 and zSeries. It includes things to look for when assessing your code to do a port, as well as differences in implementation when compared to Linux on other platforms. The topics covered in this section (presented in no particular order) are the following:

- ▶ `va_args` implementation
- ▶ Architecture-dependent code
- ▶ Assembler code
- ▶ `ptrace` & return structure
- ▶ Little endian to big endian
- ▶ Stack frame layout and linkage is specific to S/390
- ▶ `sigcontext` structure on S/390 Linux is different from Intel
- ▶ High-order bit
- ▶ Added support needed
- ▶ `proc` file system
- ▶ Supported languages
- ▶ Shared objects

`va_args` implementation

There are differences in implementation of `va_args`. These are a few notes regarding variable argument handling under Linux 390. Commonly referred to as "varargs", variable arguments can be processed under the `stdarg.h` model or the `varargs.h` model. These header files have slightly different macros for processing variable arguments. Some applications actually mix both types. In implementations of version 2.2.16 and later, it is necessary to

```
#include <varargs.h>
```

prior to any

```
#include <stdarg.h>
```

if `varargs.h` behavior is desired. This behavior is subject to change. The following discussion was derived from examples that used `varargs.h`.

Under Linux 390, a `va_list` definition is different from other platforms - it is a structure with various members.

On Linux 390, variable argument lists are processed with the `va_start(va)` macro (where `va` is a `va_list` data type). `va_start()` fills in the structure above, creating a `va_list` with all the correct members. These are needed by the system to correctly find the arguments passed in a variable argument list.

Arguments are then processed with the `va_arg` macro as follows for an integer argument:

```
varint = va_arg(va, int); /* pulls off one more argument, an integer and
                           assign to varint (an integer variable) */
```

When a `va_list` is no longer needed, the storage should be cleared with a `va_end(va)` macro (which generates an error if `va` is subsequently processed as a `va_list`).

One non-portable use of `va_lists` is the “raw” assignment of one `va_list` to another. This works on implementations where `va_list` is not implemented as a struct.

The correct way to copy one `va_list` to another `va_list` is to replace the copy with the macro:

```
__va_copy(dest, src); /* copies from src to dest */
```

When a `va_list` variable is passed as a parameter to a function, the variable is initially a pointer to `va_list` structure (because a local variable has been created and no `va_start()` has yet been performed). To access the actual arguments inside this `va_list`, the recommended approach is to either process the variable with `va_start()` or use a local variable and `__va_copy(va2, va)` to copy the contents of the passed variable into the local variable. Without these steps, either bad results or a segmentation violation is the likely outcome.

Architecture-dependent code

Programs residing in directories (on non-S/390 systems) with names like `/sysdeps` or `/arch` typically contain architecture-dependent code. You need to examine programs in these directories to ensure they are compatible with the zSeries architecture.

Assembler code

Any assembler code needs to be rewritten in S/390 Assembler. Opcodes have to be changed to S/390 opcodes, or if code uses assembler header files, you need a S/390 version of the header. S/390 Assembler code for Linux uses the 390 opcodes but follows the syntax conventions of GNU assembler. The GNU assembler manual can be downloaded at:

<http://www.gnu.org/manual/gas-2.9.1/as.html>

ptrace & return structure

The use of ptrace and the return structure is architecture-dependent.

Little endian to big endian

S/390 is a big endian system. Any code that processes data that originated on a little endian system may need some byte-swapping.

Stack frame layout and linkage is specific to S/390

See `/usr/src/linux/Documentation/Debugging390.txt` for details. Location of this file may vary depending on the distribution. In one instance, it was found in the following file:

`/usr/src/linux-2.2.16.SuSE/Documentation`

sigcontext structure on S/390 Linux is different from Intel

The PSW address at the time of interrupt and page fault (for `sigsegv`) are not stored in `sigcontext` on S/390. A patch may be developed to return this information in registers instead. Check your version of `sigcontext.h` to see if any additional fields are present that might return the address of the page that had the `sigsegv`. To find the failing PSW, the following structures in `sigcontext.h` may be used.

Figure B-1 Locating the PSW address

```
intpsw = (caddr_t)context->sregs->regs.psw.addr;

typedef struct
{
    _psw_t psw;
    unsigned long gprs[__NUM_GPRS];
    unsigned int  acrs[__NUM_ACRS];
} _s390_regs_common __attribute__((packed));

typedef struct{
    unsigned int fpc;
    double      fprs[__NUM_FPRS];
} _s390_fp_regs;

typedef struct
{
    _s390_regs_common regs;
    _s390_fp_regs      fpregs;
} _sigregs;

struct sigcontext
{
    unsigned long  oldmask[_SIGCONTEXT_NSIG_WORDS];
```

```
        _sigregs        *sregs;  
};
```

High-order bit

The high-order bit for some address fields may need to be "ANDed" off (31-bit mode bit)—for example, if the address is used in arithmetic.

Added support needed

Configuration, build, and Makefile scripts or files probably need to add support for the s/390 platform.

proc file system

The proc file system has some differences:

- ▶ /proc/cpuinfo format is different
- ▶ /proc/interrupts is not implemented
- ▶ /proc/stat does not contain INTR information

Supported languages

Programming languages for Linux on zSeries include:

- C and C++
- Perl
- Tcl
- Python
- Scheme
- Regina (Rexx)
- Java

Shared objects

Linux currently does not support shared objects like mutexes, semaphores and conditional variables across different processes.

For example, calls to:

- pthread_mutex_attr_setpshared
- pthread_rwlockattr_setpshared
- pthread_condattr_setpshared

will fail if PTHREAD_PROCESS_SHARED is set in the respective flag argument passed to these functions. When porting from platforms which support process sharing with these functions, it will be necessary to rewrite the application if the PTHREAD_PROCESS_SHARED flag is set (the Linux kernel will return a -1 if this flag is used). For mutexes, simple lock-unlock type semaphores, and condition

variables, the `pthread*` calls have to be replaced with user-written code. See `/usr/include/asm/atomic.h` for serializing functions.

Graphics support on the mainframe

If you need to load or generate an image on the server side, you might get into problems, because there is no real X server available. With Java 1.4 you can use “headless Java”¹.

However, Java 1.4 is currently not available on Linux s/390 and many libraries, including the Java AWT, require an X server. To get around this problem, you can use the following:

- Pure Java AWT, available at:

<http://www.eteks.com/pja/en/>

“PJA (Pure Java AWT) Toolkit is a Java library for drawing graphics developed by eTeks. It is 100% Pure Java and doesn't use any native graphics resource of the system on which the Java Virtual Machine runs.”

Note: The PJA consumes a lot of memory and tends to be slow.

- X Virtual Frame Buffer (Xvfb), available at:

<http://www.linuxcentral.com/linux/man-pages/Xvfb.1x.html>

“Xvfb is an X server that can run on machines with no display hardware and no physical input devices. It emulates a dumb frame buffer using virtual memory.”

Note: For Xvfb, you might want to create a start script `/etc/rc.d` similar to the following:

```
Xvfb :10&
sleep 120
export DISPLAY=:10
/opt/WebSphere/AppServer/bin/startupServer.sh
```

¹ <http://java.sun.com/j2se/1.4/docs/guide/awt/AWTChanges.html#headless>

Memory debuggers

Reference:

<http://www-1.ibm.com/servers/eserver/zseries/os/linux/ldt/whitepaper2.html>

C and C++ developers have a great deal of control over dynamic memory allocation. This freedom of control, however, can lead to significant memory management problems. These problems can cause programs to experience significant performance degradation, act unpredictably, or even crash.

Memory access errors are very difficult to find by visually inspecting the code and very rarely cause observable errors. Often, a memory access occurs only in a rare combination of circumstances; thus, memory access bugs can slip past even extensive testing, to be discovered only after deployment.

Currently on S/390 there are a number of open source tools that can locate memory management errors. These tools work by using malloc replacement code. Each tool has its own code that intercepts the calls to malloc or similar services (such as realloc) and sets up its own bookkeeping information for each memory request. In some cases, a tool, when allocating requested memory, implements memory protection schemes to catch improper memory accesses. For C programs, “Memwatch” and “Electric Fence,” used in combination, provide good coverage of potential memory errors.

Another tool called “YAMD” provides this coverage for both C and C++ programs. An attractive feature of YAMD is that it can link dynamically to the existing executable programs. (The latter need not be recompiled or rebuilt.)

Application profilers

Reference:

<http://www-1.ibm.com/servers/eserver/zseries/os/linux/ldt/profs.html>

The purpose of application profiling is to determine the behavior of an application, in the context of performance, so that analyses can be conducted. The goal of the analysis is to spot performance problems in the application. On zSeries Linux there are currently three application profilers available:

gprof	The GNU profiler that is part of the binutils component of Linux. It supports profiling of single-threaded C applications.
vprof	A visual profiler that is shipped with the SuSE distribution and supports profiling of single-threaded C and C++ applications.

cprof

An open source profiler, distributed at:

<http://www.sinenomine.net/downloads/cprof-build.php>

which supports multithreaded C and C++ applications.

These tools work in a similar manner. Here is an overview of how to use them to get profile information about your application.

1. Rebuild your application with appropriate hooks compiled into your code. This is generally done by adding compiler and/or link options when the program is built. These hooks generate calls to routines that get invoked at entry and exit to the functions you want to get profile information about.
2. Data collection. The types of data that may be collected are:
 - Call graph information
 - Time spent in each function
 - Statistical sampling
3. Post-processing of the output file to generate data in a format that can be analyzed. Once this step is complete, the application execution characteristics can be determined and improved accordingly.

Porting UNIX applications to Linux: Hints and tips

Reference:

<http://www.ibm.com/servers/eserver/zseries/library/techpapers/gm130115.html>

Then download the PDF file.

Key questions to consider before starting

Porting to Linux can be easy, simple, and straightforward, particularly if your UNIX applications are written according to common open standards. If you think that a move to Linux is attractive, you need to analyze the potential costs and risks involved in the port and how to migrate them.

Will migration involve a huge initial investment

Costs - Will the porting involve a huge initial investment of time, people and money? Will the project freeze all other new work and consume entire teams? Are big capital outlay and retraining costs required up front? Is it an all-or-nothing proposition that, once begun, can only either be completed or backed out in full?

The answer is that porting to Linux is manageable. The paper available at:

outlines how porting can be staged one step at a time, where each intermediate step is stable by itself. This gives you the freedom to manage costs, people and projects according to your priorities. It means you can commit each step independently and reassess your priorities and goals after each one. The result is a much more manageable risk, and less impact on your business.

How much will it cost, and how long will it take

To determine costs in time and money, thoroughly assess your application with respect to factors relevant to porting:

- ▶ Compiler dialect
- ▶ Hardware-dependent constructs in the code (such as word length or byte-endian)
- ▶ Platform run-time services
- ▶ Build-tool dependencies
- ▶ Availability of database, networking and messaging middleware
- ▶ User interface portability
- ▶ Test environment and test cases

The size and complexity of the porting effort varies directly in proportion to the amount of system- and environment-dependent code. If your application uses only standard language constructs and standard libraries, it may be relatively easy to port. Java applications, for example, usually fall into this category.

If, on the other hand, your application is a C program that uses non-POSIX services on Solaris, or depends on third-party products that are not available on Linux, the move can be substantially harder. Usually the system test and testing the configuration and installation of the software is an important step and makes up a major part of the port.

Will my application continue to work on the original UNIX platform

Even after moving development to Linux, you can continue to keep the original platform open in order to address your other market. Porting to Linux means that build tools are replaced by GNU tools, and POSIX-compliant threading libraries are used instead of platform-specific ones. GNU tools and libraries are both available on other UNIX platforms, allowing you to serve your original platform and Linux concurrently.

Porting from Linux to Linux on zSeries

Now that the port to Linux on one platform, say Intel, is completed, what about porting to Linux on the other hardware platforms? The short answer is: it's a piece of cake.

Why porting to Linux on zSeries?

Linux on zSeries is pure Linux. It is neither a Linux personality on an existing zSeries operating system, nor a special version of Linux adapted to the zSeries architecture. It has the same characteristics on zSeries that it would have on other platforms; for example, it is a pure ASCII environment. The vast majority of the Linux structure is common to all architectures. The zSeries-dependent modifications enable Linux to communicate with memory, disk and communications hardware. The parts of Linux which interface with applications and users are unaffected.

Application advantages

Your Linux/UNIX applications on Linux on zSeries will give you the advantage of accessing enterprise data (back-end integration) that is stored on zSeries environments. This provides improved responsiveness and reduces unnecessary duplication of data. The superior capacity, scalability, reliability and security of the zSeries makes it the perfect deployment platform for enterprise server applications. Linux on zSeries, in addition, can help to simplify operations and reduce costs by cutting the number of servers in the business environments.

Special attention should be paid to the following items in porting to Linux on zSeries:

- ▶ Little endian to big endian
- ▶ Assembler code
- ▶ Absolute addresses and high-order bit
- ▶ Application development tools

Summary

- ▶ Porting from any UNIX-to-Linux is a project that can be staged in a way that will not impact any aspect of your existing business. The complexity of porting varies from application to application and is determined by what programming language is used.
- ▶ The only skills required are common skills for application developers, so you should have all the skills you need to get started. In addition, IBM provides technical support teams and porting centers around the world, which can help you to get your applications ported.
- ▶ The detailed porting roadmaps provided in the referenced document cover valuable hints and tips for helping to judge the complexity of porting your

application. This is the base for a consistent porting plan, leading your project to a successful completion.

Solaris-to-Linux porting guide

Reference:

<http://www-106.ibm.com/developerworks/linux/library/l-solar/>

This guide is the technical version of the information presented in “Porting UNIX applications to Linux: Hints and tips” on page 282.

This discussion concerns porting from Solaris, but it may also be used for other mainstream UNIX systems. Recommended steps:

1. Download the necessary development tools and the Linux distribution.
2. Build your C/C++ application for Linux on Solaris.
 - Convert makefiles: Build your application using the GNU gmake utility instead of the Solaris make utility. There might be incompatibilities that you should resolve.
 - Compile and debug: Change the name of the C compiler from cc to gcc, and the name of the C++ compiler from CC to g++. Then recompile the code. Check the error messages. Use the compiler documentation to modify the makefiles to compensate for the differences in compiler options.
3. Become familiar with Linux while still running on Sun hardware.
4. Move the application on the target Linux platform.

Java applications

Because the JVM accepts the same bytecodes regardless of the operating system in which the JVM runs, you have the option of compiling Java source files on one operating system and running the resulting class files on another.

Fortran applications

If your Solaris Fortran application is f77-conformant, then you can use the GCC Fortran 77 (g77) compiler to compile your application code. The GNU Compiler Collection suite include the g77 compiler.

If your application is currently built with the Sun f90 or f95 compiler, then you will have to purchase a commercial Fortran 90/95 for Linux. Check the URL in the beginning of this section for details.

Run-time interfaces

Although the vast majority of run-time interfaces are common between Linux and Solaris, there are some areas where differences exist. Any use your application makes of an interface available on Solaris, but that is not available on Linux, will need to be modified before your application will build correctly. The areas of differences are listed here:

- ▶ System calls and C library
- ▶ C++ library
- ▶ Math library
- ▶ X libraries and window manager
- ▶ Desktop: CDE versus GNOME/KDE
- ▶ Thread/LWP (Light Weight Process) support
- ▶ Process management: /proc file system

Additional considerations

- ▶ System management
- ▶ Source-code management
- ▶ Other third-party tools, utilities, and libraries
- ▶ 64-bit computing
- ▶ Endian format
- ▶ Education

Technical guide for Solaris to Linux application porting

Reference:

http://www-1.ibm.com/servers/esdd/articles/porting_linux/

Of the porting guides in this appendix, this guide is by far the most comprehensive and technical.

Important: Although Solaris and Linux belong to the UNIX family, the differences between them pose a great many “gotchas”.

Porting overview

The porting strategy itself is reasonably simple:

1. Clean up the code and header files and remove architectural dependencies and nonstandard practices.
2. Compile the code and fix problems found during compile time.

3. Fix segment faults and unaligned accesses, if necessary.
4. Recompile the code and repeat the above process, if necessary.

Use the grep command

After you identify your porting development platform, you need to search for the following, which are likely to cause porting problems in the source code:

- ▶ Regular expressions
- ▶ Macros for `printf`, `sprintf`, `scanf`, and `sscanf` routines
- ▶ Structures and unions that may change the data alignment
- ▶ The `#else ... #endif` statements
- ▶ The system header files (such as `limits.h`, `types.h`, and so on)

Identify potential problems

After using the **grep** command, you also need to check for inefficient or non-portable code. The following list helps to identify the porting issues:

- ▶ Identify source code and library incompatibilities.
- ▶ Enforce stricter type-checking rules than the compiler enforces.
- ▶ Identify potential problems with variables.
- ▶ Identify potential problems with functions.
- ▶ Identify problems with flow control.
- ▶ Identify legal constructions that may produce errors or inefficiencies.
- ▶ Identify unused variable and function declarations.
- ▶ Identify unused variable and function declarations.
- ▶ Identify possibly non-portable code.

Use a porting tool

The following tools are either available or under development:

- ▶ The Porting Manager is a Perl script that takes as input a source code tree. It scans the C/C++ code for Solaris-only APIs and flags them. It also provides documentation on how you might port a Solaris API to an equivalent API for Linux. The scanning is table-driven (a table of APIs to check for and flags are provided with the tool). It also checks for include files and flags them if they are not found on your Linux system. Porting Manager has a GUI front end and, because it is written in Perl, it runs on Solaris as well as Linux and presumably anywhere that Perl runs.
- ▶ DeveloperWorks Solaris-to-Linux porting tool is a Web application that checks the APIs used by a Solaris application for compatibility on Linux. The following Web site provides access to the Web application tool:

<http://www.ibm.com/developerworks/linux/tools/1-solar.html>

- ▶ MigraTEC porting suite is from MigraTEC, a company that specializes in tools for migrating applications from one platform to another. The porting suite includes the Migration WorkBench, which provides a complete Solaris-to-Linux API map:

http://www.migratec.com/MigraTEC/migration_suite.htm

This guide presents an exhaustive list of differences between Solaris and Linux in the following areas:

- ▶ Directories
- ▶ SIGNAL handling
- ▶ Thread handling
- ▶ make
- ▶ C compiler options
- ▶ Linker options
- ▶ System-derived data types

Tools for Windows workstations

This appendix describes the following:

- ▶ Accessing Linux on zSeries using a Windows Telnet and SSH client
- ▶ UNIX emulation and XFree86 XServer running on Windows
- ▶ WebSphere Application Server.

PuTTY - a Telnet/SSH client for Windows

PuTTY is a free implementation of Telnet and SSH for Win32 platforms, along with an xterm terminal emulator. It is more efficient than the standard Telnet of Windows.

PuTTY can be obtained from:

<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

On that page select the version that is most appropriate for your workstation.

One useful component in PuTTY is pscp, a tool for copying files securely between computers using an SSH connection.

To send files to a remote server:

```
pscp [options] source [user@]host:target
```

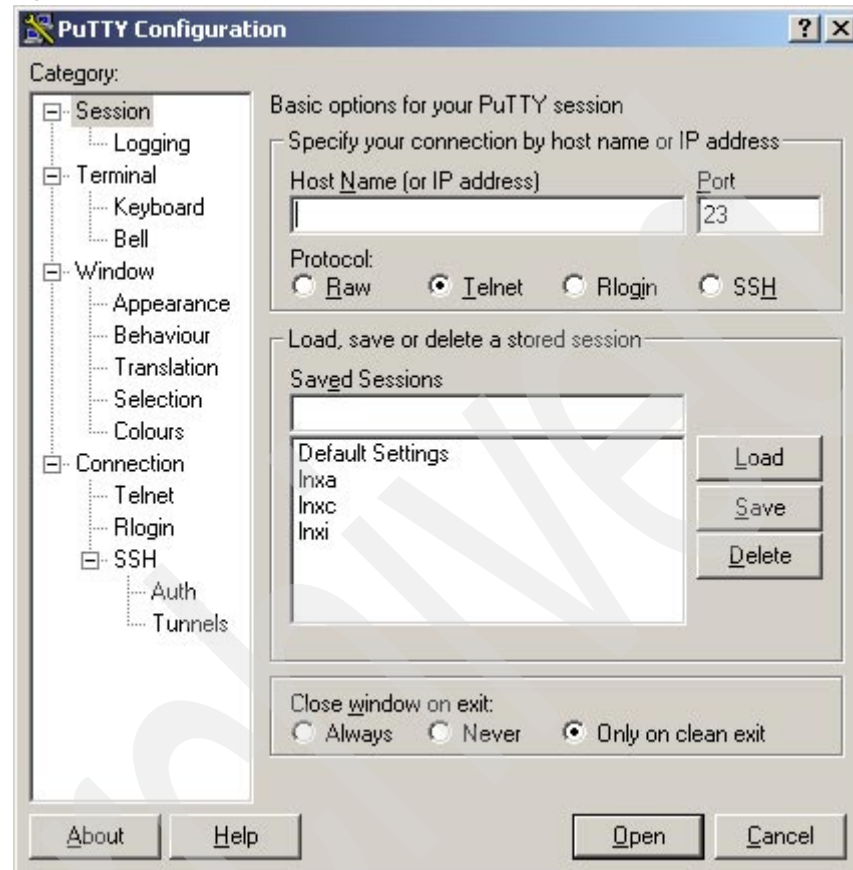
To receive files from a remote server:

```
pscp [options] [user@]host:source target
```

Note that **source** and **target** are the files in the respective computers (they may contain wildcard characters).

Figure C-1 shows a screen shot of PuTTY. For more information, click **Help** on this screen.

Figure C-1 PuTTY screen



Cygwin - a UNIX emulator for Windows

Cygwin is a UNIX/Linux environment for Windows. It consists of two parts:

1. A DLL (called `cygwin1.dll`), which acts as a UNIX emulation layer, providing substantial UNIX API functionality,
2. A collection of tools, ported from UNIX, that provide the UNIX/Linux look and feel.

Cygwin can be obtained from:

<http://www.cygwin.com/>

To install Cygwin:

1. Go to:
<http://www.cygwin.com/>
2. Click the **Install Cygwin now** button at the upper right corner to download a program (setup.exe) to some temporary directory.
3. Use Windows Explorer to find that file, then double-click it. The setup.exe will guide you further in the installation. During the process it will prompt you several times to make a choice among several options. These are the choices that we made:
 - Install from Internet.
 - Root directory: \cygwin.
 - Default Text File Type: DOS.
 - A mirror FTP site that is “closest.”
 - All packages (takes quite a while).
 - Create an icon on the Desktop.

Using XFree86 as an X Server under Cygwin

Before installing XFree86 for Cygwin, you must first install Cygwin. If you have included XFree86 in your selection while installing Cygwin, you already have XFree86 for Cygwin on your workstation! Otherwise, follow the instructions here. XFree86 under Cygwin can be obtained from:

<http://xfree86.cygwin.com/>

Installation of XFree86 is similar to that of Cygwin:

1. Go to:
<http://xfree86.cygwin.com/>
2. Click **Install Cygwin/Xfree86 now** at the upper right corner to download the setup.exe file on some temporary directory. (In fact, this file is identical to the one used to install Cygwin.)
3. Use Windows Explorer to find that file and double-click it. The setup.exe will guide you further in the installation process. These are the choices that we made:
 - Install from Internet.
 - Root directory: \cygwin.
 - Default Text File Type: DOS.
 - A mirror FTP site that is “closest”.

- XFree86 base package from the XFree86 category.
- Create an icon on the Desktop.

To start XFree86:

Click the XFree86 icon on the desktop or, if not available, enter this command from a Windows command prompt:

```
C:\cygwin\usr\X11R6\bin\startxwin.bat
```

For more information on how to use a Windows workstation as X server, refer to 14.4.1, “Graphical interface in a UNIX environment” on page 212.

Tip: Enable cut and paste between Windows and Linux

Cited from the Cygwin home page:

“Xwinclip is a clipboard manager for Cygwin/XFree86 that integrates the X Window System clipboard with the Microsoft Windows clipboard. xwinclip currently supports two-way text transfers.”

Download:

<http://xfree86.cygwin.com/devel/xwinclip>

You also need bunzip2 to extract the download. Select package **Utils->bzip2** during Cygwin setup.

Using WebSphere Application Server on Windows

Using WebSphere Application Server, it is possible to develop Java code on a Windows desktop and later deploy the application under Linux for zSeries. In this section we list some hints and tricks.

Filename

Remember that file names are case-sensitive on Linux (this is not the case on Windows).

Carriage return/line feed

If your Java editor under Windows does not support the UNIX style of saving text files, you see a carriage return (^M) at the end of every line under UNIX. To

convert text files into a UNIX text file, you can use Jakarta Ant. Example C-2 shows how an Ant target was used to automate the process.

Figure C-2 Converting DOS-style carriage return to UNIX-style using Ant

```
<target name="fixcrlf">
  <fixcrlf srcdir="." includes="**/*.java, **/*.jsp, **/*.properties" />
</target>
```

For reference, see:

<http://jakarta.apache.org/ant/manual/CoreTasks/fixcrlf.html>

Property files under WebSphere Studio Application Developer

Do *not* store *.properties files under *webapp*/WEB-INF/classes in WebSphere Studio Application Developer. (For example, the log4j manual tells you to put the log4j.properties file in the /WEB-INF/classes directory. Do *not* do it!) We make this recommendation for the following reasons:

- ▶ WebSphere Studio Application Developer deletes this directory and regenerates the content.
- ▶ If you want to put a property file in the CLASSPATH, store it under the source directory. WebSphere Studio Application Developer will copy it to the *webapp*/WEB-INF/classes directory.

Command line completion under Windows

For command line completion under Windows (similar to the bash shell on Linux), start *regedit.exe* and modify the following key:

```
HKEY_CURRENT_USER\Software\Microsoft\Command Processor: CompletionChar=9
```

Copying files from Windows to Linux

You can copy files between Windows and Linux in several ways.

WebDav

Using WebDav, Linux directories served by Tomcat can be mapped as a network drive on Windows. To enable it:

1. Enable write access in the */var/tomcat4/webapps/webdav*/WEB-INF/web.xml configuration file (see the *<init-param>* stanza).
2. Use Windows Explorer to map the directory as a network drive:
 - a. Navigate to **Tools -> Map Network Drive**.

b. Create a shortcut to the directory:

```
location=http://server:8180/webdav
```

PutTTY

Use the secure copy that is available with PuTTY - the **pscp** command. Here is an example usage:

```
pscp -r -pw "password" *.jar user@10.1.1.128:/tmp
```

FTP

After enabling ftp on the server, you can use any ftp client to upload and download files (your Netscape browser, for example):

```
ftp://USERNAME:PASSWORD@10.1.1.128/tmp
```

Samba

You can map a directory on the Linux server as a network drive on your workstation. Detailed instruction on installation and configuration are available at:

<http://www.tldp.org/HOWTO/SMB-HOWTO.html>

Java Virtual Machine environment

Make sure you have the same environment variable for the Java Locale on Windows and Linux. For example:

```
export LC_ALL=en_US
```

Just in Time compiler (JIT)

If your code runs on Windows and you get strange messages when running under Linux, try disabling the JIT compiler.

For JDK 1.1, issue:

```
java -nojit
```

For JDK 1.2, issue:

```
java -Djava.compiler=None
```


Linux for S/390 VM HONE pilot

This appendix describes a pilot project conducted in the EMEA HONE/LINK platform (EMEA stands for Europe, Middle East and Africa). The purpose of this pilot project was to deliver a demonstration Linux for S/390 e-business solution under the slogan “Doing business with Linux on the mainframe”.

The pilot encompassed:

- ▶ Installation of a Linux for S/390 server on the S/390 VM platform, where IBM's worldwide HONE/LINK service is running
- ▶ Installation of WebSphere, IHS, DB2 on this Linux server
- ▶ Development of JavaServer Pages and backend connection servlets in the Linux WebSphere environment to enable three real HONE/LINK business applications

The selected applications have all been successfully enabled in the new environment. The development of this solution for production rollout of the HONE/LINK Web application portfolio has been approved and is underway.

The pilot can be reached at:

<http://lwebx01.linux.ehone.ibm.com>

Introduction

Note: This appendix is extracted, with permission, from a white paper titled “Doing business with Linux on mainframe - Linux for S/390 VM HONE Pilot” by Richard van Wel, IBM Netherlands.

Unlike the other chapters of the redbook, in this appendix the terms *we* and *us* refer to the Architecture & Technology Innovation and the HONE/LINK development groups in IBM Netherlands.

IBM is currently committed to making a large scale investment in Linux (ref [1]). As part of this there is much interest in the progress of Linux on the mainframe (zSeries, S/390). The prospect of *virtualizing* hundreds or even thousands of Linux servers on a mainframe offers spectacular opportunities for cost reduction and time saving in comparison to conventional Webserver farms which require a physical server for each Linux image. There is a definite trend underway for the mainframe to be in the IT spotlight once again.

The expectation is that Linux will be the pre-eminent platform for the next generation e-business applications. An estimated 80% of enterprise business data resides on mainframes. In view of its mainframe market share IBM holds the trump card with the combination of Linux and the mainframe.

In early 2001 the AMS Architecture & Technology Innovation and the HONE/LINK Development teams jointly decided to conduct a Linux mainframe pilot, combining mainframe experience, new Linux S/390 technology and e-business in order to demonstrate the feasibility of Linux mainframe e-business solution. This would make use of the S/390 VM platform known as HONE/LINK that has traditionally supported a number of important business applications.

A virtual Linux server on this platform serves as a Web portal to three key HONE/LINK legacy applications that run on the same platform. The Linux server is equipped with standard WebSphere components, which support the developed Java solution (servlets, JSPs) which, in turn, exploits an XML-based “legacy connector” to access the legacy applications.

The architectural overview diagram Figure D-1 illustrates the key points of the solution.

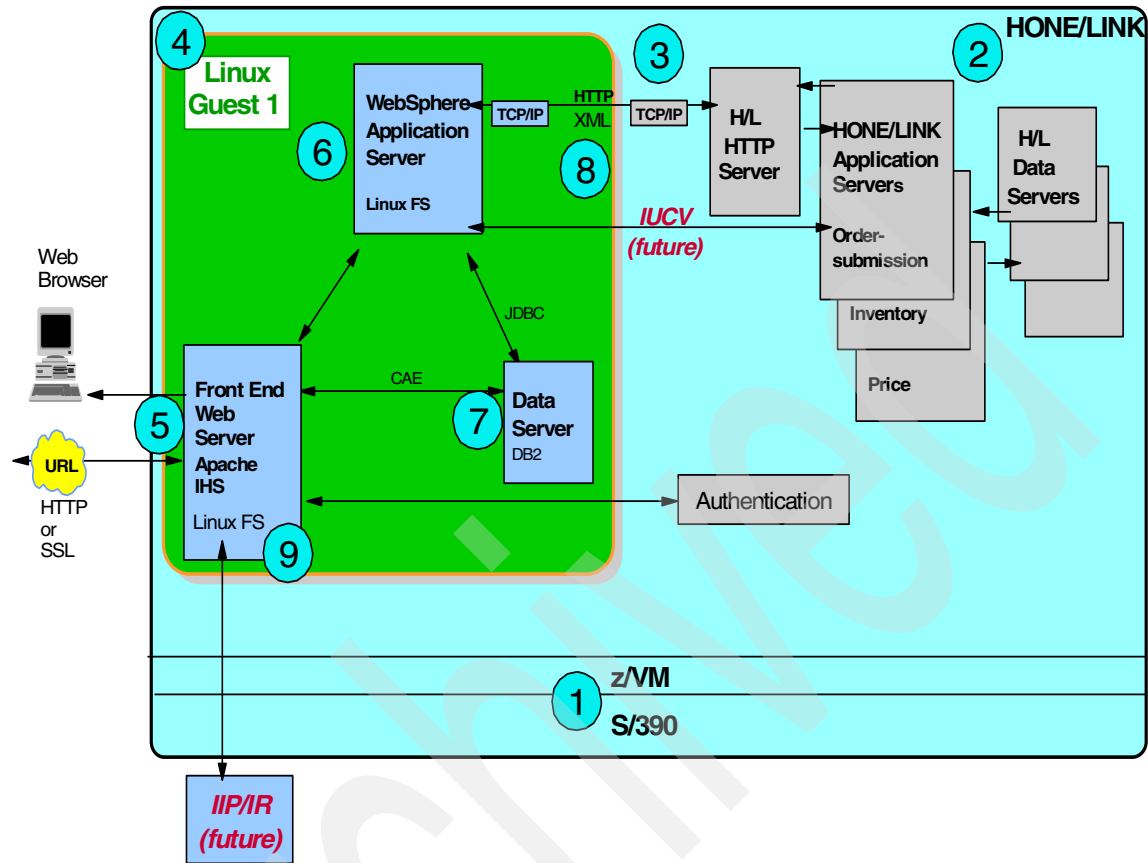


Figure D-1 HONE Pilot Architectural overview

The zSeries (S/390) mainframe is running under the z/VM operating system (1). This is the platform on which the HONE/LINK legacy applications (2) are running. HTTP access to these applications is enabled through a VM HTTP server (3). The pilot project introduces a Linux server (4) running as a guest under VM with IBM's standard WebSphere middleware installed: the IBM HTTP Server (5) serving the user's Web browser requests, the WebSphere Application Server (6) running JSPs and Java servlets and the DB2 Data Server (7) handling WebSphere's administration. One of the servlets connects the Linux Web serving code via HTTP to the back end applications (8). The HONE/LINK applications require their users to be authenticated against the legacy security repository (RACF) which is done via an IHS plug-in (9).

Two future improvements to this setup are shown.

- ▶ The connection via HTTP to the back-end applications (8) is to be replaced by a direct IUCV in-memory connection which allows to bypass the HONE/LINK HTTP server (3) and is expected to boost performance.
- ▶ The authentication via the IHS plug-in (9) is to be taken over by IBM Intranet Password (IIP) or IBM Registration (IR), with either IIP or IR userid being mapped to the legacy userid for application authorization.

Background

In this chapter we

- ▶ briefly introduce the HONE/LINK service
- ▶ describe its evolution from legacy green screens to Web enablement
- ▶ discuss GWA compliance issues
- ▶ present the Linux for S/390 opportunity
- ▶ discuss the pilot and the production rollout project.

HONE/LINK

HONE/LINK is the name of a service which started in 1975 to support Marketing and Services. Over time it has supported IBM hardware and software configurators, product information, technical support and ordering. Today ordering has remained as the most important set of applications. Order submission, Inventory retrieval and Price query etc. are the key components of IBM's supply chain in EMEA.

After its start in the United Kingdom in 1975, the mission which is originally named "Hands-On Networking Environment" (HONE) was moved to the Netherlands in 1977. In 1982 it was decentralized to many countries ("country HONE").

In 1987 access was extended to external users ("IBMLink"), hence the name change to HONE/LINK: HONE for IBM internals, LINK for externals.

In 1994 the service was centralized again in EMEA, followed by worldwide consolidation in two sites: one in Germany for EMEA, the other in the US for the US. The other geographies are evenly divided between the two sites. This is still the situation today.

HONE/LINK is hosted on zSeries mainframes running under the z/VM operating system. In 1988 the conventional green screens were extended with Web

interfaces for the most important applications (“HONWeb” and “LINKWeb”). In 2001 the next step in this evolution was taken with the pilot of Linux front end which is the subject of this Appendix.

Some figures are in order to give an idea of the importance of the HONE/LINK platform. On average 100,000 sessions are executed monthly on HONWeb/LINKWeb (that is, via the Intranet/Internet Web interface), There are still another 150,000 sessions each month executed in the old-fashioned green screen mode. In 2001 a total of 190,000 orders were placed through this channel by IBM Sales and Business Partners, worth over \$5B of IBM's revenue stream.

Green screens

For a good understanding of the evolution of a typical legacy mainframe system like HONE/LINK to the e-business platform of today, let us have a look first at the green screen situation that still in use today. Figure D-2 illustrates the setup.

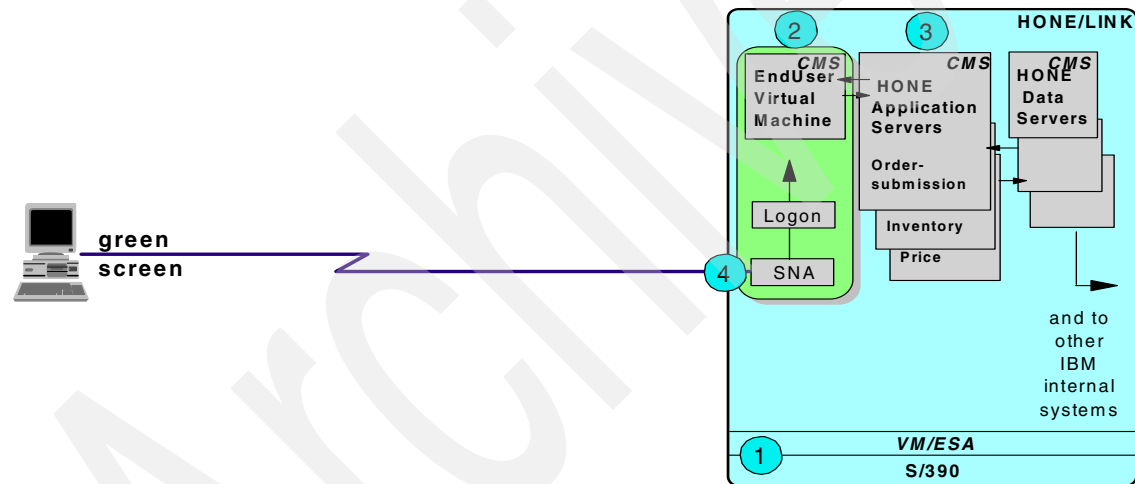


Figure D-2 Legacy green screen setup

On the S/390 hardware platform of the VM/ESA (called z/VM in newer releases) operating system is in control (1). This is an operating system which allows the virtual partitioning of hardware between other operating systems (e.g. z/OS and recently Linux) running on this same hardware, giving these operating system the impression that they are in control of the hardware. However, the commands of their supervisors to the hardware are intercepted by the VM operating system which does the actual mapping. As it apparently resides above these

supervisors, it is sometimes called a “hypervisor”. In order to sustain a good level of performance this mapping utilizes hardware assistance. Also, much gain is obtained through clever “handshaking” with the operating systems - this has improved z/OS (MVS) performance under VM considerably over the years. Undoubtedly Linux under z/VM will go through a similar maturing process.

The Conversational Monitor System (CMS) is an operating system which has been developed to run solely under VM - it cannot even run without VM. It can be viewed as a “virtual PC” of which each user gets a personal copy. A user gets access to his virtual machine (2) by logging on to VM. After authentication of userid and password CMS is started and the user gets a piece of storage, a few cylinders of hard disk and access to some of the processors’ capacity.

In this virtual machine he can develop and run his own programs but, more importantly, he can also link his virtual machine to other virtual machines and run their programs. Most interesting here are “server virtual machines” (3) which are running continuously without a human user logged on and which provide application functions of general interest.

Finally the green screens that the user gets presented have originally been designed for display via SNA, IBM’s proprietary Systems Network Architecture protocol on 3270 based display stations (4).

HONEWeb/LINKWeb

Web enabling of the HONE/LINK legacy applications is illustrated in Figure D-3 on page 303 which is a fair representation of today’s production environment.

The user virtual machines (Figure D-2 on page 301 (2)) here have been replaced by CMS server virtual machines (1) which listen to the VM TCP/IP HTTP ports and interact with the legacy applications in the same way as the user virtual machines did before. These Web server virtual machines run Common Gateway Interface (CGI) programs that communicate with the user’s Web browser via HTTP and HTML forms (2).

There is no more need to log on, the URL is all that is needed to connect to the HONEWeb or LINKWeb home page. However, most applications need user identification, in which the user will still be prompted for his legacy userid and password that will be authenticated against the legacy security repository (3) as before. The HONE/LINK Web servers support HTTP Basic authentication.

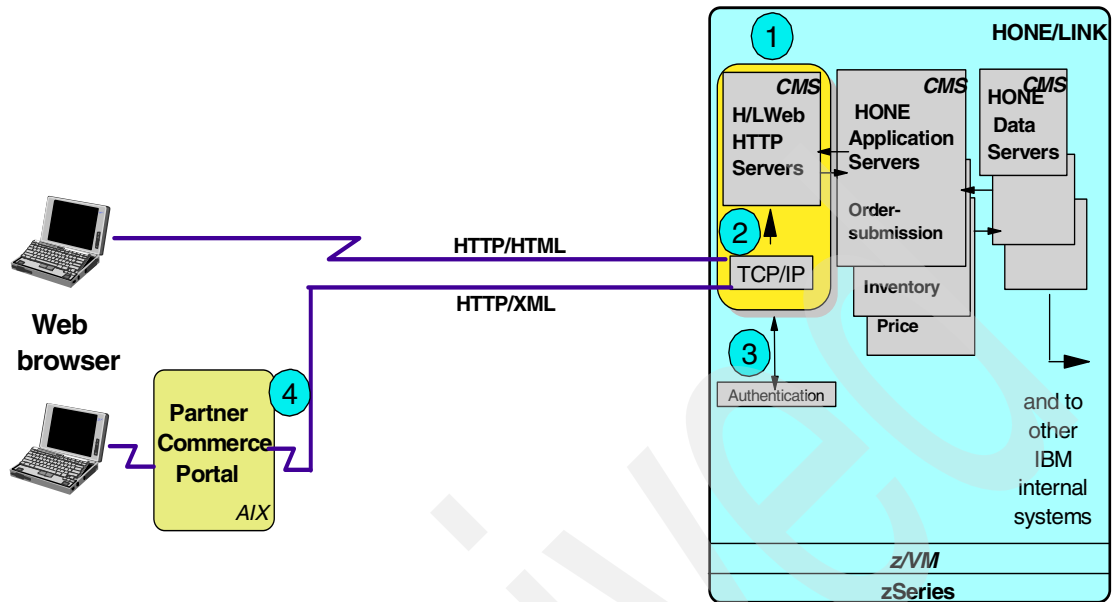


Figure D-3 HONE/LINK current implementation

So far this supports user browser-to-program communication. The next step is where the user at the calling end is replaced by another Web site, e.g. the PartnerCommerce portal site (4), which requires program-to-program communication. The calling program does not want HTML responses, it just wants the data back, so it can build its own style pages to return to the end users. HONE/LINK also supports this communication mode and uses XML to tag the returned data. Conceptually the HONE/LINK applications have been turned here into “Web Services”. The use of the Simple Object Access Protocol (SOAP) will be the logical next step.

GWA

The use of Web servers under VM is not supported in IBM internal Global Web Architecture (GWA). Several alternatives have been considered, see Figure D-4.

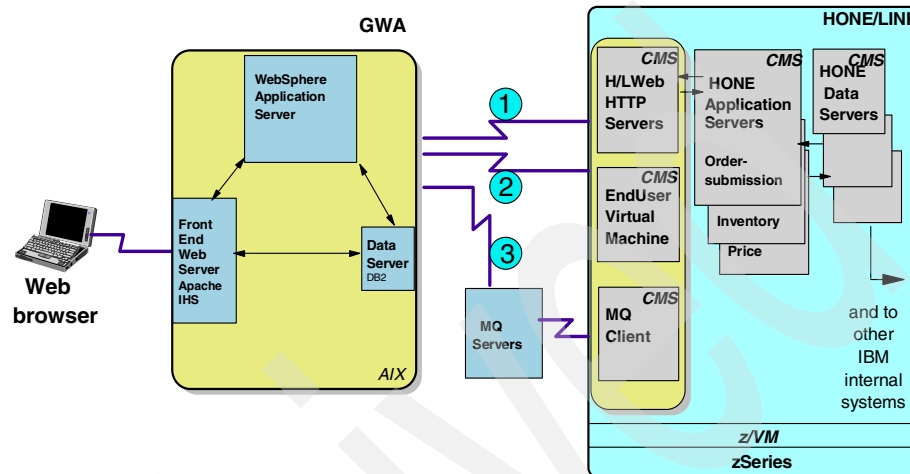


Figure D-4 HONE/LINK alternative implementation

All alternatives take a GWA compliant AIX system as starting point.

- ▶ Alternative (1) is the situation just described between the PartnerCommerce portal and HONE/LINK (refer to Figure D-3 on page 303 (4) where the GWA system connects to the legacy system via HTTP. This still requires the VM HTTP servers which are only used here for basic communication, not for Web page serving.
- ▶ Alternative (2) requires the use of HostPublisher product on the GWA system. HostPublisher logs on via telnet in the same way as the old green screen user, i.e. with userid and password, to the user virtual machine. Indeed, green screens are actually returned to the HostPublisher, and the “screen scraping” method is used to parse the data and reuse these, so that the GWA system can build its pages from this input.

A few years ago IBM advocated the use of HostPublisher company-wide in a massive attempt to migrate all VM legacy applications to GWA. This failed due to the complexity of most legacy screens which makes screen scraping a hazardous undertaking.

- ▶ Alternative (3) makes use of MQSeries between the GWA system and HONE/LINK. Due to its asynchronous nature this is unsuitable for applications that operate in real time with end users, as is the case with HONE/LINK. Also, VM only supports MQ Client which implies that the queries must be hosted

elsewhere and that remote queue management programs have to be developed on the VM side.

Linux for S/390

The defined goal of the pilot was to Web-enable a few selected key legacy business applications with Linux on the S/390 VM HONE platform. The standard IBM WebSphere middleware should be used, coding should be entirely in Java (servlets, JSPs) and VisualAge for Java should be used.

The pilot took place started on March 2001 and ended on October 2001 when the three selected applications were fully functioning.

Production rollout

For the production rollout some additional applications will be ported to the new environment. Furthermore, some more work in tuning and configuring are needed to improve performance.

A significant performance improvement can be obtained by replacing the interface between the Linux server and application backend server via the VM HTTP servers (Figure Figure D-1 on page 299) by a direct in-memory IUCV connection.

Work has started on enhancing the existing low-level Linux IUCV driver with an API to make this happen. This code may be of enough general interest to be submitted to the Open Source community.

System configuration

The configuration of the pilot system is shown in Table D-1.

Table D-1 HONE pilot system configuration

Hardware	ELINK12 - IBM 9672-R76 7-way, CMOS G5 technology 713 MIPS 2GB Main Storage 14GB Expanded Storage
Host Operating System	IBM z/VM V3.1 SLU 0101
Guest Configuration	V=V 2-way 512MB Main Storage
Guest Operating System	SuSE Linux 7.0 for S/390 - Kernel 2.2.16

Hardware	ELINK12 - IBM 9672-R76 7-way, CMOS G5 technology 713 MIPS 2GB Main Storage 14GB Expanded Storage
WebSphere	WebSphere Application Server Advanced Edition V.3.5.3 for Linux for S/390
httpd (Web server)	IBM HTTP Server V.1.3.12

The production system will use SuSE SLES-7 (S390) - Kernel 2.4.7-SuSE-SMP and WebSphere Application Server Advanced Edition V4.0 for zSeries.

Conclusion

The Linux for S/390 has proven to be an economical solution. With little cost we have been able to set up a Linux server with WebSphere, IHS and DB2 and migrate three legacy applications to this new environment. The low cost is mainly because we make use of existing hardware and do not have to modify the legacy applications

At the same time we have brought the next generation e-business platform in place, upgraded our technical skills and delivered a showcase for IBM's Linux strategy.

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246807>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246807.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
sg246807.tar.gz	Code Samples

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 500 KB minimum

Operating System: Linux for IBM

How to use the Web material

Create a subdirectory (folder) on your server, and unzip the contents of the Web material zip file into this folder.

Abbreviations and acronyms

ANT	Another Neat Tool	SWT	Standard Widget Toolkit
CDK	C Development Kit	TLD	Tag library descriptor
CDT	C/C++ Development Kit	WAR	Web Application Archive
CVS	Concurrent Versions System		
EJB	Enterprise Java Beans		
FAQ	Frequently Asked Questions		
GUI	Graphical User Interface		
GWA	Global Web Architecture		
HONE	Hands On Network Environment		
HTML	HyperText Markup Language		
IBM	International Business Machines Corporation		
IDE	Integrated Development Environment		
IHS	IBM HTTP Server		
ITSO	International Technical Support Organization		
J2SDK	Java 2 Software Development Kit		
JAR	Java Archive		
JDK	Java Development Kit		
JDT	Java Development Toolkit		
JNDI	Java Naming Directory Interface		
JRE	Java Runtime Environment		
JSDK	Java Servlet Development Kit		
JSP	JavaServer Pages		
MVC	Model-View-Controller		
PDE	Plug-in Development Kit		
Releng	Release Engineering		
RPM	RedHat Package Manager		
SDK	Software Development Kit		

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 315.

- ▶ *WebSphere Version 4 Application Development Handbook*, SG24-6134
- ▶ *Linux for WebSphere and DB2 Servers*, SG24-5850
- ▶ *Linux on IBM @server zSeries and S/390: ISP/ASP Solutions*, SG24-6299

Other resources

These publications are also relevant as further information sources:

- ▶ *z/Architecture Principles of Operation*, SA22-7832
- ▶ Michael K. Johnson and Erik W. Troan, *Linux Application Development*, Addison-Wesley Publishing, ISBN 0-20130-821-5
- ▶ *IBM DB2 Universal Database Version 7 Application Development Guide*, SC09-2949
- ▶ Karl Fogel, *Open Source Development with CVS*, CoriolisOpen Press, 1999, ISBN 1-57610-490-7
- ▶ Ian F. Darwin, *Java Cookbook*, O'Reilly & Associates Inc., 2001, ISBN 0-596-00170-3
- ▶ Linda Lam and Arnold Robbins, *Learning the Vi Editor*, O'Reilly & Associates Inc., 1998, ISBN 1-56592-426-6
- ▶ Gamma et al., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1998, ISBN 0-201-63361-2
- ▶ *Version Management with CVS* by Per Cederqvist et al., found at:
<http://www.cvshome.org/docs/manual/>
- ▶ *The Linux Programmer's Guide* by Sven Goldt, Sven van der Meer, Scott Burkett, and Matt Welsh found at:
<http://www.linuxhq.com/guides/LPG/>

- ▶ *A Look at the Signal API* by Erik Troan, found at:
http://www.linux-mag.com/2000-01/compile_01.html

Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ Linux for IBM @server zSeries home page
<http://www.ibm.com/servers/eserver/zseries/os/linux/>
- ▶ Linux for IBM @server zSeries application tools page
http://www.ibm.com/servers/eserver/zseries/os/linux/ldt/slate_enablers.html
- ▶ SpeakEasy.Rpfind.net server
<http://www.rpfind.net/>
- ▶ Sun Microsystem Java home page
<http://java.sun.com/>
- ▶ IBM Developer Kit for Java home page
<http://www.ibm.com/developerworks/java/jdk/>
- ▶ IBM Developer Kit for Java for Linux (Version 1.3)
<http://www.ibm.com/developerworks/java/jdk/linux130/>
- ▶ The Jikes Java compiler home page
<http://www.ibm.com/developerworks/oss/jikes/>
- ▶ The Emacs reference page
<http://www.stanford.edu/group/dcg/leland-docs/emacs.html>
- ▶ The Hessling editor home page
<http://hessling-editor.sourceforge.net/>
- ▶ The Jakarta project home page
<http://jakarta.apache.org/>
- ▶ The Jakarta Tomcat server version 4 documentation page
<http://jakarta.apache.com/tomcat/tomcat-4.0-doc/index.html>
- ▶ The Jakarta Tomcat server version 4.0.3 RPM distribution page
<http://jakarta.apache.com/builds/jakarta-tomcat-4.0/release/v4.0.3/rpms/>
- ▶ The Jakarta Tomcat server version 4 configuration documentation page
<http://jakarta.apache.com/tomcat/tomcat-4.0-doc/config/index.html>

- ▶ The Jakarta Tomcat server version 4 application manager documentation
<http://jakarta.apache.com/tomcat/tomcat-4.0-doc/manager-howto.html>
- ▶ The Jakarta Ant version 1.4.1 binary distribution
<http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/bin/>
- ▶ The Jakarta Ant reference manual
<http://jakarta.apache.org/ant/manual>
- ▶ The Jakarta Log4j download page
<http://jakarta.apache.org/log4j/docs/download.html>
- ▶ The Jakarta Taglib home page
<http://jakarta.apache.org/taglibs/index.html>
- ▶ The Jakarta Struts documentation page
<http://jakarta.apache.org/struts/doc-1.0.2/>
- ▶ The Jakarta Struts API documentation page
<http://jakarta.apache.org/struts/doc-1.0.2/api/>
- ▶ The Jakarta Struts download page
<http://jakarta.apache.org/builds/jakarta-struts/release/v1.0.2/>
- ▶ Linux for S/390 - Notes and observations
<http://linuxvm.org/penguinvm/notes.html>
- ▶ IBM zVM Internet library
<http://www.vm.ibm.com/library/>
- ▶ The GNU profiler (gprof) manual
http://sources.redhat.com/binutils/docs-2.10/gprof_toc.html
- ▶ Using an porting the GNU compiler collection (gcc) page
<http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc.html>
- ▶ The Eclipse project home page
<http://www.eclipse.org/>
- ▶ The Sun Java tutorial on Internationalization page
<http://java.sun.com/docs/books/tutorial/i18n/index.html>
- ▶ The Sun Java JDBC page
<http://java.sun.com/products/jdbc/>
- ▶ The Sun Java servlet specification page
<http://java.sun.com/products/servlet/download.html>

- ▶ The Electric Fence download page
<ftp://ftp.perens.com/pub/ElectricFence/>
- ▶ The Linux Documentation Project page
<http://www.tldp.org/docs.html>
- ▶ The Linux Documentation Project RPM HOWTO page
<http://www.tldp.org/HOWTO/RPM-HOWTO/>
- ▶ The Qt reference documentation page
<http://doc.trolltech.com/3.0/index.html>
- ▶ The POSIX specification
<http://opengroup.org/onlinepubs/007904975/basedefs/contents.html>
- ▶ The POSIX specification for pthread_barrier_wait
http://opengroup.org/onlinepubs/007904975/functions/pthread_barrier_wait.html
- ▶ The Sun Java servlet specification page
<http://java.sun.com/products/servlet/download.html>
- ▶ Database pooling with WebSphere Application Server
<http://www-3.ibm.com/software/webservers/studio/appserver40pooling.html>
- ▶ The Linux Documentation Project RPM HOWTO page
<http://www.tldp.org/HOWTO/RPM-HOWTO/>
- ▶ WebSphere Application Server V4.0 trial download page
<http://www14.software.ibm.com/webapp/download/search.jsp?go=y&rs=wasael>
- ▶ The DB2 UDB trial download page
<http://www6.software.ibm.com/d1/db2udbd1/db2udbd1-p/>
- ▶ The GNU assembler documentation page
<http://www.gnu.org/manual/gas-2.9.1/as.html>
- ▶ The Pure Java AWT (PJA) toolkit
<http://www.eteks.com/pja/en/>
- ▶ The virtual framebuffer X server (Xvfb) man page
<http://www.linuxcentral.com/linux/man-pages/Xvfb.1x.html>
- ▶ The Sun Java AWT enhancements for headless displays page
<http://java.sun.com/j2se/1.4/docs/guide/awt/AWTChanges.html#headless>
- ▶ Tools for application profiling on Linux for zSeries
<http://www-1.ibm.com/servers/eserver/zseries/os/linux/ldt/profs.html>

- ▶ Building cprof on Linux/390
<http://www.sinenomine.net/downloads/cprof-build.php>
- ▶ Porting UNIX applications to Linux - hints and tips
<http://www.ibm.com/servers/eserver/zseries/library/techpapers/gm130115.html>
- ▶ Solaris-to-Linux porting guide
<http://www-106.ibm.com/developerworks/linux/library/l-solar/>
- ▶ Technical guide for porting applications from Solaris to Linux
http://www-1.ibm.com/servers/esdd/articles/porting_linux/
- ▶ MigraTEC porting suite
http://www.migratec.com/MigraTEC/migration_suite.htm
- ▶ The PuTTY telnet/ssh client home page
<http://www.chiark.greenend.org.uk/~sgtatham/putty/>
- ▶ The Cygwin project home page
<http://www.cygwin.com/>
- ▶ The XFree86 for Cygwin home page
<http://xfree86.cygwin.com/>
- ▶ Linux in IBM HONE/LINK
<http://lwebx01.linux.ehone.ibm.com/>

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

A

- Ant 80
 - build.xml 80
 - checking library dependencies 260
 - exec rule 192
 - installing 80
 - optional package 80
 - projecthelp 81
 - reference manual 81
- application profilers 281
 - cprof 282
 - gprof 281
 - vprof 281
- apropos 5
- ar 198

B

- byte ordering 90
 - big-endian 90

C

- CDT 147
 - adding extensions 156
 - compiling 150
 - debugging 151
 - first plug-in 153
 - navigating code 149
 - packaging and managing projects 152
 - running code 150
 - sample project 147
- cross-compiler 9
 - compiling binutils 10
 - compiling the compiler 12
 - installing 13
 - installing libraries 10
 - prerequisites 9
 - using 13
- CVS 33
 - adding files and directories 45
 - administration 38
 - administrative files 39
 - arguments 37

- check out 34
- command options 37
- command syntax 35
- commands 36
- commit 34
- committing changes 46
- date formats 38
- editing the working copy 44
- environment variables 40
- file locking 35
- file permissions 39
- global options 36
- import 41
- log 51
- log messages 38
- manual 33
- module 34
- obtaining a working copy 42
- remote access 39
- repository 34
- resolving conflicts 51
- revision number 34
- revision numbering 35
- root directory 39
- special files 43
- ssh access 40
- update 34
- updating the working copy 48
- working copy 34
- Cygwin 291
- XFree86 292

D

- data types 90
 - alignment 90
 - sizes 90
 - unions 91
- DB2 267
 - accessing remote databases 272
 - configuring 272
 - db2samp 272
 - db2setup 269
 - installing 268

- prerequisites 268
- debugging under zVM 102
 - cp trace 102
- diff 19
 - recursive comparison 22
- diff3 19
- dynamically linked libraries 203
 - checking for errors 205
 - include files 205
 - opening 204
 - releasing 205
 - symbol lookup 205

E

- Eclipse 109
 - concepts 132
 - CVS 125
 - editors 128
 - getting started 114
 - installing 115
 - Java Development Toolkit 113
 - Perspective 132
 - Plug-in Development Environment 113
 - prerequisite software 116
 - rebuilding 117
 - Resources 132
 - Standard Widget Toolkit 112
 - starting 124
 - Update 112
 - User Interface 112
 - Version Control Mechanism 112
 - View 132
 - WebDav 112
 - Workbench 132
- emacs 55
 - basic commands 57
 - buffers 58
 - cancel function 58
 - compiling applications 63
 - copy and paste 60
 - editing multiple files 58
 - editing program files 62
 - emerge-files 19
 - fundamental mode 61
 - GNU Emacs Reference Card 57
 - help 61
 - incremental search backward 61
 - incremental search forward 61

- indentation 62
- invoking Lisp functions 57
- kill line 60
- modes 61
- moving text 59
- multiple windows 59
- regions 60
- search and replace 61
- selecting text 59
- starting 56
- syntax highlighting 62
- the mark 60
- yank 60

F

- function calling convention 93
 - prolog and epilog 95
 - register usage 93
 - stack layout 94

G

- gcc 6
 - architecture dependent options 103
 - cross-compiler 9
 - debugging 8
 - dependences 17
 - directory search 7
 - documentation 104
 - inline functions 103
 - macros 8
 - optimization 9
 - optimization reference 104
 - performance options 102
 - source files 6
 - stages 7
 - starting 6
 - string operations 104
 - unrolled loops 103
 - warnings 8
- gdb 96
 - attach 99
 - break 99
 - breakpoints 99
 - core file 98
 - detach 99
 - dir 100
 - display 101
 - examining variables 99

- ignore 100
- including debugging information 96
- info break 100
- info stack 99
- info threads 99
- kill 99
- list 100
- print 100
- run 99
- scope operator 99
- show directories 100
- thread 99
- thread apply 99
- watch 100
- gprof 102
- Graphical User Interface 211
- graphics support 280
 - Pure Java AWT 280
 - Xvfb 280

I
info 5

J

- Jakarta 75
 - Ant 80
 - Log4J 81
 - Struts 84
 - Taglibs 82
 - Tomcat 76
- Java
 - IBM Java Developer Kit for Linux 28
 - Jikes 29
 - JIT compiler 295
- Java 2 Platform
 - Software Development Kit 28
- JDBC 180
 - connection pooling 184
 - drivers 187
 - implementation 182
- JDT 134
 - debugging 140
 - initialization 135
 - integrating with Ant 142
 - integrating with CVS 144
 - menu bar and tool bar 134
 - projects 135
 - running an application 139

- jikes 29
- JNI 188
 - implementing in C 189
 - shared library 191
 - using JNI in Java code 188

L

- ldconfig 201
- ldd 202
- Libraries 18, 196
 - dynamically linked 196
 - finding 19
 - inspecting 197
 - preparing 196
 - shared 196
 - standard C and C++ libraries 18
 - static 196
- Linking 13
- Loader 201
- locking using files 230
 - fcntl 230
 - flock 230
 - lockf 230
- Log4J
 - installing 81

M

- maintaining portability 92
- make 14
 - Makefile 14
 - target conventions 16
- makedepend 17
- Makefile 14
 - comments 15
 - rules 15
 - variables 15
- man 4
- Man pages 4
- memory debuggers 281
 - Electric Fence 281
 - memwatch 281
 - YAMD 281
- memory mapped files 207
 - logging 211
 - synchronizing memory and disk 210

N

nm 197

O

objdump 197

P

patch 21

- context patch 22
- distributing patches 24

PDE 153

- setting up the environment 153

porting applications 275

- architecture dependent code 277
- assembler code 277
- costs 282
- Fortran applications 285
- from Linux to Linux on zSeries 284
- hi-order bit 279
- Java applications 285
- little endian to big endian 278
- porting tools 287
- proc file system 279
- ptrace & return structure 278
- run-time interfaces 286
- shared objects 279
- sigcontext structure 278
- Solaris to Linux porting guide 285
- stack frame layout and linkage 278
- supported languages 279
- va_args 276

ptrace 101

PuTTY 290

- pscp 290

R

Redbooks Web site 315

- Contact us xvii

revision numbering 35

RPM 255

- building 257
- installing 258
- preparing source archive 256

S

Samba 295

sample 252

connection type 263

customizing 255

database 254

JDBC configuration 262

JNI configuration 262

prerequisite libraries 254

source structure 252

semaphores 230

ipcrm 231

ipcs 230

setrlimit 93

shared libraries 199

building 202

environmental variables 202

loading 201

names 201

shared object dependencies 202

using 200

shmat 93

signals 104

zSeries exceptions 105

source 252

spinlocks 231

creating 232

operations 232

using 233

zSeries specific features 231

SQL 238

assigning and freeing contexts 246

bind file 240

binding 238

client server considerations 249

connection context 242

context operations 242

creating a package 238

db2dclgn 240

incorporating prep/bind into make 240

multiple connections 241

package 238

sqlAttachToCtx 247

sqlBeginCtx 245

sqlDetachFromCtx 248

sqlEndCtx 249

sqlSetTypeCtx 245

static SQL files in libraries 241

static libraries 198

creating 198

preparing object files 198

using 199

- strace 101
- Struts 84
 - Action 84
 - ActionForm 85
 - ActionForward 84
 - ActionMapping 84
 - ActionServlet 84
 - configuration 178
 - configuring ActionServlet 179
 - installing 87
 - registering ActionForm 178
 - registering ActionMapping and ActionForward 179
 - struts-html 86
 - struts-bean 86
 - struts-logic 86
 - struts-template 86
 - taglibs 86
- svc 92

T

- Taglibs 82
 - configuring 82
 - installing Java classes 82
 - JSP pages 83
 - Tag Library Descriptors 82
 - web application descriptor 83
- threads 219
 - a problem 234
 - barrier 233
 - cancellation points 222
 - cleanup stack 223
 - conditional variables 228
 - creating threads 221
 - mutexes 227
 - signalling conditions 229
 - synchronizing threads 226
 - thread attributes 224
 - thread cancellation 222
 - thread stack size 224
 - thread termination 222
 - using 220
 - waiting for conditions 229
- Tomcat 76
 - application manager 79
 - configuration file 77
 - configuring 78
 - deploying applications 79

- installing 76
- port 78
- starting and stopping 77
- userid and group 76
- verifying installation 78
- WebDav 294

U

- ulimit 93

V

- va_args 276
- vi 65
 - commands 68
 - customizing 68
 - insertion point 69
 - modes 66
 - moving the cursor 68
- virtual address space 92

W

- WAR 258
 - deploying 262
 - using Ant 259
- Windows workstations 293
 - command line completion 294
 - copying files 294
 - filenames 293
 - Java Virtual Machine 295

X

- X windows 212
 - DISPLAY 212
 - QT library 213
 - X-client 212
 - X-server 212



Linux on IBM® Server Series and S/390: Application Development

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages



Redbooks

Linux on IBM ^eserver zSeries and S/390: Application Development

Tools and techniques for Linux application development

Using the Eclipse IDE and Jakarta Project tools

Sample code to illustrate programming techniques

This IBM Redbook describes application development for Linux on the IBM ^eserver zSeries platform. The target audience is application developers writing primarily in C/C++ and Java.

The Linux development environment for zSeries is quite similar to the development environment on other platforms running Linux since the operating system services and development tools share a common code base. We note differences and optimizations specific to the zSeries platform where applicable. The zSeries platform offers unique advantages to Linux application developers. Running Linux images as guests under zVM allows consolidation of development servers onto a centrally managed machine, thus simplifying system administration of the development environment. The hardware virtualization provided by zVM allows physical resources to be shared among multiple Linux guests.

In part one, we discuss standard development tools available for Linux on the zSeries platform. We provide complete details for using the IBM Java Software Development Toolkit, CVS, Emacs, the vi editor, and applications that make up the Jakarta Project.

In part two, the open source Eclipse IDE is introduced. We describe the basic concepts it incorporates, and provide step-by-step instructions for installing, configuring, and working with Eclipse.

In part three, we demonstrate programming techniques using an example J2EE application as an illustration. All the code necessary to implement the sample project in your own environment is included.

An appendix provides details about installing DB2 for Linux on zSeries, presents topics related to porting applications, and describes a pilot project that produced an e-business solution on Linux for S/390.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks