Build Guide

Sun Java™ Wireless Client Software 2.1

Java Platform, Micro Edition

Sun Microsystems, Inc.
www.sun.com

April 2008

Copyright © 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.sun.com/patents and one or more additional patents or pending patent applications in the U.S. and in other countries.

THIS PRODUCT CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, HotSpot, J2ME, J2SE, J2EE, Java Developer Connection, Java Community Process, JCP, Javadoc, JDK, JavaCall, Java Card, phoneME and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

The PostScript logo is a registered trademark of Adobe Systems, Incorporated.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, États-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs des brevets américains listés à l'adresse http://www.sun.com/patents et un ou plusieurs brevets supplémentaires ou les applications de brevet en attente aux États - Unis et dans d'autres pays.

CE PRODUIT CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS L AUTORISATION EXPRESSE, ECRITE ET PREALABLE DE SUN MICROSYSTEMS, INC.

Droits du gouvernement des États-Unis - logiciel commercial. Les droits des utilisateur du gouvernement des États-Unis sont soumis aux termes de la licence standard Sun Microsystems et aux conditions appliquées de la FAR et de ces compléments.

Cette distribution peut inclure des elements développés par des tiers.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, HotSpot, J2ME, J2SE, J2EE, Java Developer Connection, Java Community Process, JCP, Javadoc, JDK, JavaCall, Java Card, phoneME et le logo Java Coffee Cup sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux États-Unis et dans d'autres pays.

UNIX est une marque déposée aux États-Unis et dans d'autres pays et sous licence exclusive de X/Open Company, Ltd.

Intel est une marque déposée de Intel Corporation ou de sa filiale aux États-Unis et dans d'autres pays.

OpenGL est une marque déposée de Silicon Graphics, Inc.

Le logo PostScript est une marque de fabrique ou une marque déposée de Adobe Systems, Incorporated.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont régis par la législation américaine en matière de contrôlé des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manièré non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régi par la législation américaine en matière de contrôlé des exportations et la liste de ressortissants spécifiquement désignes, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DÉCLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITÉ MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIÈRE OU A L'ABSENCE DE CONTREFAÇON.

Contents

Preface xi

1. Introduction 1

Directory Structure 2

Tools 2

Environment Variables and Build Options 3

2. Quick Start: Building on Windows for Windows 5

Setting Up on Windows 5

Setting Up Your Environment 6

Setting the PATH 7

Verifying Windows Setup 8

Setting Up For the Build 9

Building the JavaCall API 10

Building PCSL 12

Building CLDC 13

Building Sun Java Wireless Client Software 14

Running Sun Java Wireless Client Software 16

Moving Sun Java Wireless Client Directories 16

Running in the WTK Emulator 17

3. Quick Start: Building on Linux for ARM 19

Setting Up Your Environment 20

Building PCSL 21

Building CLDC 23

Building Java Wireless Client Software 24

Running Java Wireless Client Software 25

4. JavaCall Build System 27

JavaCall Build Overview 27

JavaCall Build Variables 27

Extending the Build System 28

JavaCall Directory Structure 28

5. PCSL Build System 29

PCSL Build Overview 29

Output 30

Debugging Symbols 30

API Documentation 30

Selecting Modules 31

About Stubs 32

Building Individual Services 33

Network Service 33

Running Unit Tests 34

Extending the Build System 34

Creating a New Platform Makefile 34

Creating New OS and Compiler Makefiles 35

Creating a New Module 36

6. CLDC Build System 37

CLDC Build Overview 37

CLDC Build System Variables 38

7. Java Wireless Client Software Build System 39

Overview 39

Output 40

Debugging Symbols 41

API Documentation 41

Build Options 42

CLDC Selection 42

Mapping Configuration Variables 42

Module Selection 43

Native AMS Image Resource Policy 44

Multitasking 44

Startup Performance 44

Resource Allocation Policy 44

Cryptography Selection 45

Server Socket Selection 45

Runtime Java Platform Properties Selection 45

Specifying a Target CPU and Device 46

Build Constraints 46

Building Optional Package APIs 47

About Optional JSRs 47

Optional Package JSRs and Other Build Components 48

Optional API Variable Pairs 48

Using the `JSR` Variable Pair 49

Using the `ABSTRACTIONS` Variable Pair 49

Using the `make` Command Line 50

Using the `make` Command Line with Makefiles 51

Optional Package API Details 51

Building JSR 120 and JSR 205 51
Building JSR 135 and JSR 234 52
The USE_JPEG Variable 52
Building JSR 226 and JSR 172 52
Building JSR 229 and JSR 120 53
Building JSR 177 53
Building JSR 239 54
Working With Stubs 55
Configuring the Build System for Stubs 55
Updating the Build System for Filled-in Stub Functions 56
Updating the Source Files and Build System after Porting 56

Glossary 57

Index 61

Figures

FIGURE 2-1 Simulated Device 17

FIGURE 3-1 Application Manager 26

Tables

TABLE 4-1 JavaCall Build Targets 27

TABLE 5-1 PCSL Build Targets 29

TABLE 5-2 PCSL Module Selection 32

TABLE 6-1 CLDC Build System Variables 38

TABLE 7-1 Java Wireless Client Software Build Targets 40

TABLE 7-2 Debugging Selection Options 41

TABLE 7-3 CLDC Selection Options 42

TABLE 7-4 Configuration Options Mapping Between Build Systems 43

TABLE 7-5 Module Selection Options 43

TABLE 7-6 SSL Selection Options 45

TABLE 7-7 APDU Build Options 54

Preface

This book describes how to create build Sun Java™ Wireless Client Software from its source code. This guide assumes that the product is installed on a system that meets the hardware and software requirements described in the *Release Notes*.

Before You Read This Guide

Readers using this guide must be familiar with the *MIDP 2.0 Specification*.

How This Book Is Organized

Chapter 1 describes the build environment.

The next two chapters are designed to help you hit the ground running, so you can quickly build and run the Sun Java Wireless Client software.

Chapter 2 illustrates how to build and run on Windows.

Chapter 3 shows how to build on Linux to run on an ARM device.

The last four chapters in this book provide a more thorough and systematic description of the various pieces of the Sun Java Wireless Client software build system.

Chapter 4 describes the JavaCall porting layer directory structure and how to build the JavaCall porting layer.

Chapter 5 describes how to build the Portable Common Services Layer (PCSL).

Chapter 6 describes how to build the Connected Limited Device Configuration HotSpot™ Implementation.

Chapter 7 describes how to build the Sun Java Wireless Client software.

Operating System Commands

This document does not contain information on basic commands and procedures such as opening a terminal window, changing directories, and setting environment variables. See the software documentation that you received with your system for this information.

Shell Prompts

Shell	Prompt
Bourne shell and Korn shell	`$`
Windows	*directory*>

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Related Documentation

The following documentation is included with this release of the Sun Java Wireless Client software.

Application	Title
All	*Release Notes*
Porting	*Porting User's Guide*
Using Tools	*Tools Guide*
Using Adaptive User Interface Technology	*Skin Author's Guide*
Using Multitasking Features	*Multitasking Guide*
Viewing reference documentation created by the Javadoc™ tool	*Java API Reference*
Viewing reference documentation created by the Doxygen tool	*Native API Reference*

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document, and does not endorse and is not responsible or liable for any content, advertising, products, or other materials available through such sites.

Accessing Sun Documentation Online

The Java Developer Connection™ program web site enables you to access Java platform technical documentation at `http://java.sun.com/`.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. Provide feedback at
`http://java.sun.com/docs/forms/sendusmail.html`.

1

Introduction

This book describes how to build Java Wireless Client software. Java Wireless Client software has a powerful and flexible build system that supports building *on* multiple platforms *for* multiple platforms. The platform where you run Java Wireless Client software is the *target* platform.

This book describes the build system as a whole and provides several example builds. The examples are a good way to become acquainted with the Java Wireless Client software and build system.

Once you are comfortable building Java Wireless Client software, you can begin to adapt the build to your own specific hardware.

Building a complete Java Wireless Client software stack consists of several steps. Basically, you start at the bottom and work your way up.

1. Build the JavaCall™ API.
2. Build PCSL.
3. Build CLDC.
4. Build Java Wireless Client software and optional APIs.

Each build uses the results of previous steps.

This chapter provides a brief overview of the directory structure of the Java Wireless Client software. It also lists the tools you need to build the software.

This book describes scripts that you can modify and use to build the Java Wireless Client software. These scripts are available for your use in the `docs/build-scripts` directory.

Directory Structure

The top level of the Java Wireless Client software contains `javacall`, `javacall_com`, `pcsl`, and `cldc` directories that correspond to the JavaCall API, PCSL, and CLDC parts of the stack. The source code and makefiles for each layer are contained in the corresponding directory.

The rest of the Java Wireless Client software stack, MIDP and optional APIs, is contained in the rest of the top level directories. For example, `midp` contains the source files and makefiles for the MIDP APIs. The `jsr135` directory contains the source code for JSR 135 Mobile Media API, and so on.

The Java Wireless Client software build is driven by the makefiles in `midp`. By setting flags for the build system, you can include source code for optional APIs in the Java Wireless Client software build.

Tools

The exact set of tools that you need to build the Java Wireless Client software depends upon your target platform and the optional APIs that you plan to include.

In general, however, you need the following:

1. GNU Make
2. UNIX® system-style tools
3. A C/C++ compiler
4. The Java Platform, Standard Edition (Java SE platform) Development Kit (JDK™) version 1.4.2 (not 1.5.x)

Consult the *Release Notes* for more specific information and recommendations for tool versions.

Upcoming chapters will discuss how to set up and verify your tools for specific types of builds.

Environment Variables and Build Options

Always specify build options on the `make` command line. In some cases, it is possible to set environment variables to supply values to the build system, but it is a bad idea.

For example, suppose you did something like this:

```
export SOME_OPTION=true
```

```
make
```

If `SOME_OPTION` has no definition in the build system, the environment variable's value is used. On the other hand, if `SOME_OPTION` is defined with a default value in the build system, the value of the environment variable is ignored.

Mechanisms that work some times and not other times only lead to pain and frustration.

Instead, define all build options on the `make` command line, like this:

```
make SOME_OPTION=true
```

This always works as you expect.

For more information about using `make`, see "Using the `make` Command Line" on page 50.

2

Quick Start: Building on Windows for Windows

This chapter describes building on Windows for Windows. It covers building and running your very own Java Wireless Client software stack. The purpose of this chapter is to whisk you through the build process for the entire Java Wireless Client software stack. Later chapters describe all the options for each part of the build.

This chapter presents example scripts that help you set up your environment and run the different parts of the build. The scripts are available in `$HOME/docs/build-scripts/win32-emul`.

Once you have everything set up, run the scripts to perform the build. Work along as you read through the chapter to understand how it all fits together.

Setting Up on Windows

Consult the *Release Notes* for the very latest information on tool versions.

The core of the Java Wireless Client software build on Windows is Microsoft's compiler, `cl.exe`. Use Microsoft Visual C++ 6.0 Professional Edition. Although other development packages from Microsoft include the compiler, they unsupported.

To use the compiler, you need to add it to your `PATH` environment variable and update the `INCLUDE` and `LIB` variables to appropriate values. Microsoft provides a batch file to set `PATH`, `INCLUDE`, and `LIB` to appropriate values. This batch file is `VCVARS32.BAT`.

GNU Make and other UNIX system-style tools are available for Windows in a package called Cyg4Me. Get it here:

ftp://ftp.sunfreeware.com/pub/freeware/contributions/cygwin/cyg4me1_1_full.zip

Cyg4Me is a specialized version of a more widely known package, Cygwin. Use Cyg4Me rather than Cygwin.

Note – If you already have experience with Cygwin, the temptation will be to continue to use it. Although Cygwin and Cyg4me are similar, they not the same. Cyg4me is the preferred tool set for building the Sun Java Wireless Client software.

Setting Up Your Environment

The initial setup of your build environment is best done from the Cyg4me directory where you have installed the Cyg4me software, for example, in the directory `D:\cyg4me`.

1. **Change directory to** `$HOME/docs/build-scripts/win32-emul`**.**
2. **Copy the file** `setpath.bat` **to your Cyg4me location.**
3. **Copy the remainder of the setup scripts to the top-level of the Sun Java Wireless Client software distribution, for example,** `D:\jwc`**. These scripts include:**
   - `setup.sh`
   - `teardown.sh`
   - `build-javacall.sh`
   - `build-pcsl.sh`
   - `build-cldc.sh`
   - `build-sjwc.sh`
4. **Change directory to the location of your Cyg4me installation and follow the instructions in** “Setting the PATH” on page 7**.**

Setting the PATH

You will need to set your path so that all of the tools are available. Here is an example batch file.

```
@echo off

REM Path to vcvars
set VCVARS=d:\PROGRA~1\MIAF9D~1\VC98\Bin\VCVARS.BAT

REM JDK with backward slashes.
set JDK_DIR_win32=d:\j2sdk1.4.2_13

REM JDK with forward slashes (used in later scripts).
set JDK_DIR=d:/j2sdk1.4.2_13

REM Cyg4Me
set CYG4ME=d:\cyg4me

REM *********************************************

REM Set up PATH, INCLUDE, and LIB for C++ compiler.
call %VCVARS%

REM Add JDK.
set PATH=%JDK_DIR_win32%\bin;%PATH%

REM Add Cyg4me as the first PATH element.
set PATH=%CYG4ME%\bin;%PATH%
```

This batch file is `setpath.bat`. Edit it so the values for the following variables are correct for your build system:

- `VCVARS` points to the `VCVARS32.BAT` file provided with Microsoft Visual C++.
- `JDK_DIR_win32` is the JDK directory with backward slashes, Windows-style.
- `JDK_DIR` is the JDK directory with forward slashes.
- `CYG4ME` points to your installation of Cyg4Me.

After you set the values appropriately, run the batch file like this:

```
D:\cyg4me>setpath
Setting environment for using Microsoft Visual C++ x86 tools.
D:\cyg4me>
```

Run the scripts in the rest of this chapter from the same Windows command line.

The Windows build is sensitive with respect to paths. Be careful when you define them, as certain forms of paths succeed in some parts of the build and fail in others. Follow the supplied scripts as closely as possible until you are comfortable with the build system.

Note – The examples in this book are based on having the entire source code tree in the default Sun Java Wireless Client software installation directory on your chosen drive, for example, `D:\jwc`. In some cases, Windows may have problems with extended path names. If you run into trouble, you may need to shorten your paths. If you do this, be careful and make sure all scripts and makefiles take into account the modified paths.

Verifying Windows Setup

These examples show how to find out if you have set your PATH correctly and have the right tools available:

- **GNU Make**

```
D:\cyg4me>make --version
GNU Make version 3.78.1, by Richard Stallman and Roland McGrath.
Built for Windows32
Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99
    Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR
A PARTICULAR PURPOSE.

Report bugs to <bug-make@gnu.org>.
```

- **C/C++ compiler**

```
D:\cyg4me>cl
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version
    14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option ] filename [ /link linkoption ]
```

- **Macro Assembler**

```
D:\cyg4me>ml
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corporation 1981-2000.  All rights
reserved.

usage: ml [ options ] filelist [ /link linkoption ]
```

-

- **JDK software**. Verify that java and javac are both from the same location using which. Don't be confused by the cygdrive designation, just make sure that which reports the same directory for java and javac.

```
D:\cyg4me>java -version
java version "1.4.2_13"
Java(TM) 2 Runtime Environment, Standard Edition
    (build 1.4.2_13-b06)
Java HotSpot(TM) Client VM (build 1.4.2_13-b06, mixed mode)
Usage: javac <options> <source files>
where possible options include:
...
D:\cyg4me>which java
/cygdrive/d/j2sdk1.4.2_13/bin/java

D:\cyg4me>which javac
/cygdrive/d/j2sdk1.4.2_13/bin/javac
```

Setting Up For the Build

The remainder of the build scripts presented in this chapter operate using Cyg4Me's shell, not the Windows command line. However, you will execute the scripts from the Windows command line, using sh to invoke the Cyg4Me shell. You'll see examples as you read the rest of the chapter.

Each of the build scripts in the rest of this chapter operate by setting up, doing some work, and cleaning up.

The setting up script is setup.sh and the cleaning up script is teardown.sh.

You have previously copied the build scripts to the top-level directory of y our Sun Java Wireless Client software distribution, for example, D:\jwc.

The `setup.sh` script looks like this:

```
export HOME=d:/jwc
export COMPONENTS_DIR=$HOME
export Scripts=`pwd`
export Output=$HOME/output
export Log=$HOME/log.txt
rm -f $Log
```

The `HOME` variable is the top level of the Java Wireless Client software. If needed, adjust its value to point to the top level of your own installation. The `HOME` variable is then assigned to the variable `COMPONENTS_DIR`, which is required by the Sun Java Wireless Client software build makefiles.

There is no need to run `setup.sh`; it is called by the other scripts.

`teardown.sh` looks like this:

```
cat $Log
cd $Scripts
```

You do not need to modify the `teardown.sh` script.

Building the JavaCall API

Three variables must be defined to build the JavaCall API.

Two of these variables point to the JavaCall API source code, which is split into a base (open source) component and a product (commercial) component. Use `JAVACALL_DIR` to point to the base JavaCall API directory, usually `javacall`. Next, point `PROJECT_JAVACALL_DIR` to the `javacall-com` directory.

`JAVACALL_OUTPUT_DIR` tells the build system where to put the output of the JavaCall API build.

`TOOLS_DIR` and `TOOLS_OUTPUT_DIR` define the location of needed build tools and a location for tools-specific output.

Run the build from the top-level Sun Java Wireless Client software location, for example, D:`\jwc`.

Here is a simple script to build the JavaCall API, build-javacall.sh:

```
. setup.sh

make -C $HOME/javacall-com/configuration/sjwc/win32_emul \
JAVACALL_DIR=$HOME/javacall \
PROJECT_JAVACALL_DIR=$HOME/javacall-com \
JAVACALL_OUTPUT_DIR=$Output/javacall \
TOOLS_DIR=$HOME/tools \
TOOLS_OUTPUT_DIR=$Output/tools \
$1

if [ $? -ne 0 ]; then
  echo "javacall_status=failed" >> $Log
else
  echo "javacall_status=OK" >> $Log
fi

. $Scripts/teardown.sh
```

You should be able to run this script unmodified. This script uses the setup.sh and teardown.sh scripts from the previous section.

Go ahead and give it a whirl. Remember, you are still using the Windows command line, but you're going to invoke the script using Cyg4Me's sh shell. Messages display as the build progresses, with a success message at the very end.

```
D:\jwc>sh build-javacall.sh
.
.
.
javautil_jad_parser.c
javautil_string.c
javautil_unicode.c
...Generating Library: d:/jwc/output/javacall/lib/javacall.lib
...compiling resources ...
d:\Applications\cyg4me\bin\make.exe:
javacall_status=OK

D:\jwc>
```

The output goes in $HOME/output/javacall. Take a look:

```
D:\jwc>ls ./output/javacall
ext_lib  inc  lib  obj

D:\jwc>
```

Building PCSL

The next step is to build PCSL. You have to tell PCSL the target platform and where to put output files. Because you are building PCSL on top of the JavaCall API, which is built on top of Windows (i386) using Visual C++ (vc), the PCSL_PLATFORM variable should be javacall_i386_vc.

JAVACALL_OUTPUT_DIR needs to be set so that the PCSL build can find the JavaCall API files that it needs. Usually, JAVACALL_OUTPUT_DIR will still be set from your JavaCall API build.

The third variable, PCSL_OUTPUT_DIR, tells PCSL where to put its output files.

The fourth and fifth variables, TOOLS_DIR and TOOLS_OUTPUT_DIR, define the location of needed build tools and a location for tools-specific output.

Here is a script, build-pcsl.sh, that performs the build:

```
. setup.sh

make -C $HOME/pcsl \
JAVACALL_OUTPUT_DIR=$Output/javacall \
PCSL_PLATFORM=javacall_i386_vc \
PCSL_OUTPUT_DIR=$Output/pcsl \
TOOLS_DIR=$HOME/tools \
TOOLS_OUTPUT_DIR=$Output/tools \
$1

if [ $? -ne 0 ]; then
  echo "build-pcsl=failed" >> $Log
else
  echo "build-pcsl=OK" >> $Log
fi

. $Scripts/teardown.sh
```

This script uses the same setup.sh and teardown.sh scripts as before. Again, you can use build-pcsl.sh without modifying it.

Run the script the same way you ran the JavaCall API build script:

```
D:\jwc>sh build-pcsl.sh
.
.
.
building memory module...
d:\cyg4me\bin\make.exe[4]: Entering directory
`d:/jwc/pcsl/memory/heap'
d:\cyg4me\bin\make.exe[4]: Leaving directory
`d:/jwc/pcsl/memory/heap'
d:\cyg4me\bin\make.exe[3]: Leaving directory `d:/jwc/pcsl/memory'
d:\cyg4me\bin\make.exe[2]: Leaving directory
`d:/jwc/pcsl/string/utf16'
d:\cyg4me\bin\make.exe[1]: Leaving directory
`d:/jwc/pcsl/string'
build-pcsl=OK

D:\jwc>
```

At the end is a message about the success of the PCSL build.

Browse through the output files in `output/pcsl` if you want to see the results of the build.

Building CLDC

CLDC is built on top of PCSL and the JavaCall API. Tell CLDC to use PCSL by setting `ENABLE_PCSL` to `true`, and specify where to find PCSL with `PCSL_OUTPUT_DIR`.

Tell the CLDC build system where to find the JavaCall API with `JAVACALL_OUTPUT_DIR`

Use `TOOLS_DIR` and `TOOLS_OUTPUT_DIR` to define the location of needed build tools and a location for tools-specific output.

In addition, the CLDC build system expects you to define several environment variables:

- `JDK_DIR` is the location of the Java Development Kit software.
- `ENABLE_PCSL` tells the build to make use of PCSL components.
- `PCSL_OUTPUT_DIR` is where the CLDC build script looks for PCSL.
- `JVMWorkSpace` is the `cldc` directory.

- The output of the CLDC build is placed in `JVMBuildSpace`.
- `Use` `TOOLS_DIR` to define the location of the build tools.
- Use `TOOLS_OUTPUT_DIR` define a location for tools-specific output.

The following script, `build-cldc.sh`, runs the CLDC build.

```
. setup.sh

make -C $HOME/cldc/build/javacall_i386_vc \
JDK_DIR=$JDK_DIR \
ENABLE_PCSL=true \
PCSL_OUTPUT_DIR=$Output/pcsl \
JAVACALL_OUTPUT_DIR=$Output/javacall \
JVMWorkSpace=$HOME/cldc \
JVMBuildSpace=$Output/cldc \
TOOLS_DIR=$HOME/tools \
TOOLS_OUTPUT_DIR=$Output/tools \
$1

if [ $? -ne 0 ]; then
  echo "cldc_status=failed" >> $Log
else
  echo "cldc_status=OK" >> $Log
fi

. $Scripts/teardown.sh
```

Run the script using the command:

`D:\jwc>sh build-cldc.sh`

After a few minutes and a success message at the end of the build, you can browse the output in `output/cldc`.

Building Sun Java Wireless Client Software

The last step in creating a Java Wireless Client software stack is building MIDP and optional APIs. This chapter describes how to build MIDP only. Later chapters describe how to include optional APIs in this part of the build and describe the many variables you can use to change how Java Wireless Client software is built.

For now, build MIDP using the CLDC, PCSL, and the JavaCall API output you already generated.

You need JDK_DIR defined, just as it was for the CLDC build. In addition, you must tell the MIDP build where to find the JavaCall API, PCSL, and CLDC:

- Define JAVACALL_PLATFORM as win32_i386_vc and use JAVACALL_OUTPUT_DIR to point to the output of the JavaCall API build.
- Set PCSL_OUTPUT_DIR as shown in "Building CLDC" on page 13.
- Set CLDC_DIST_DIR to the dist directory of the CLDC build you wish to use. In this case, it will be $Output/cldc/javacall_i386_vc/dist.
- Use TOOLS_DIR to define the location of the tools directory.
- Use TOOLS_OUTPUT_DIR define a location for tools-specific output.

Finally, set MIDP_OUTPUT_DIR to the location that contains the output of the build when it is complete.

Run the build from the top-level Sun Java Wireless Client software directory, for example, D:\jwc. Here is a script, build-sjwc.sh, that sets the necessary variables and runs the MIDP build:

```
 . setup.sh

make -C $HOME/midp/build/javacall \
JDK_DIR=$JDK_DIR \
JAVACALL_PLATFORM=win32_i386_vc \
JAVACALL_OUTPUT_DIR=$Output/javacall \
PCSL_OUTPUT_DIR=$Output/pcsl \
CLDC_DIST_DIR=$Output/cldc/javacall_i386_vc/dist \
TOOLS_DIR=$HOME/tools \
TOOLS_OUTPUT_DIR=$Output/tools \
MIDP_OUTPUT_DIR=$Output/midp \
$1

if [ $? -ne 0 ]; then
  echo "sjwc_status=failed" >> $Log
else
  echo "sjwc_status=OK" >> $Log
fi

. $Scripts/teardown.sh
```

Run the script using the command:

```
D:\jwc>sh build-sjwc.sh
```

Wait until you see the message sjwc_status=OK at the end of the build. Congratulations! You've just built a complete MIDP stack.

Running Sun Java Wireless Client Software

The Sun Java Wireless Client software target environment is the Wireless Toolkit (WTK) running on a Windows 2000 or Windows XP platform. Therefore, in order to run successfully in the WTK environment, some additional steps are needed after you built the source.

Moving Sun Java Wireless Client Directories

When you have finished building the Sun Java Wireless Client, you will see the following directories in your top-level build output directory (the location you specified in the build steps with the variable `$MIDP_OUTPUT_DIR`):

- `/WTK`
- `/WTK_STORAGE`

The contents of these two directories must be moved to appropriate locations in your Wireless Toolkit environment. Once they are moved, the Sun Java Wireless Client software can be used seamlessly with the Wireless Toolkit.

Note – Following these instructions assumes a default Wireless Toolkit configuration and a `DefaultColorPhone` setting for the emulator. If your environment has been reconfigured or you are using some other device, you may encounter difficulties in getting set up.

Move the contents of the two Sun Java Wireless Client software directories (not the directories themselves) to the following WTK locations:

- `/WTK` - Goes in your top-level WTK directory, for example, `C:\WTK2.5.2`.
- `/WTK_STORAGE` - Goes in your default Windows home directory, for example, `C:\Documents and Settings\`<*user_name*>`\j2mewtk\2.5.2\appdb`

Note – The first part of the above path may be set on your Windows platform as a system variable, `%USERPROFILE%=C:\Documents and settings\`<*username*>.

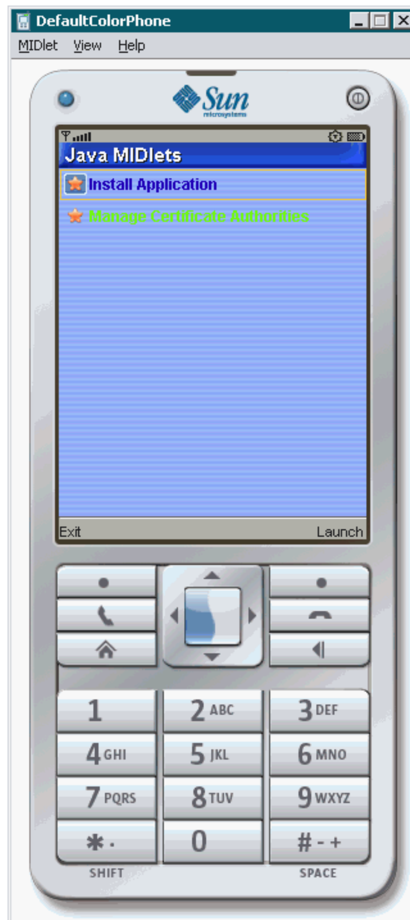
Once these Sun Java Wireless Client software files have been moved, no other set up or additional configuration should be needed.

Running in the WTK Emulator

Once you have properly set up the Sun Java Wireless Client to run in the Wireless Toolkit emulator, you interact with the emulator in the same way you would using any other MIDlet or MIDlet suite.

You can install and test MIDlets on this simulated device, as shown in FIGURE 2-1. For details, see the *Sun Java Wireless Client Tools Guide* or the Wireless Toolkit documentation.

FIGURE 2-1 Simulated Device



3

Quick Start: Building on Linux for ARM

This chapter is about *cross-compiling*, where you build the Java Wireless Client software on a Linux desktop computer but run it on a device with an ARM processor.

Linux builds come in two flavors:

- Framebuffer (`fb`) builds use a simple virtual display. You can use the `qvfb` tool to see and interact with the Java Wireless Client software.

This chapter describes how to perform the `linux_fb_gcc` build for an ARM processor.

This chapter is based on example scripts that help you run the different parts of the build. The scripts are available in `jwc/docs/build-scripts/linux-arm`. Once you have everything set up, run the scripts to perform the build. Work along as you read through the chapter to understand how it all fits together.

Setting Up Your Environment

Most of the time, Linux already has most of the tools you need. These examples show how to find out if you have these tools available. If you do not, you will need to install the appropriate tools and adjust your `PATH` to include them.

To check for GNU Make, execute the following command:

```
$ make --version

   GNU Make 3.80
   Copyright (C) 2002  Free Software Foundation, Inc.
   This is free software; see the source for copying conditions.
   There is NO warranty; not even for MERCHANTABILITY or FITNESS
   FOR A PARTICULAR PURPOSE.

$
```

On your computer, GNU Make might be `gmake` instead of `make`.

To check for the GCC cross compiler, execute the following command:

```
$ gcc --version

   3.3.1

$
```

To check for JDK software, execute the following command:

```
$ java -version

   java version "1.4.2_13"
   Java(TM) 2 Runtime Environment, Standard Edition
      (build 1.4.2_13-b03)
   Java HotSpot(TM) Client VM (build 1.4.2_13-b03, mixed mode)
   $ javac
   Usage: javac <options> <source files>
   where possible options include:
   ...
$
```

Each of the build scripts in the rest of this chapter operate by setting up, doing some work, and cleaning up.

The setting up script is `setup.sh` and the cleaning up script is `teardown.sh`.

setup.sh looks like this:

```
export HOME=/jwc
export JDK_DIR=/usr/java/j2sdk1.4.2_13
export GNU_TOOLS_DIR=/opt/Embedix/tools/arm-linux
export Scripts=`pwd`
export Output=$HOME/output
export Log=$HOME/log.txt
rm -f $Log
```

This script is available as jwc/docs/build-scripts/linux-arm/setup.sh.

The HOME variable is the top level of the Java Wireless Client software. Adjust the value to point to the top level of your own installation. Also, make sure JDK_DIR points to the top directory of your JDK. Finally, adjust GNU_TOOLS_DIR to point to the installation directory of your GCC cross compiler.

Don't run setup.sh yet. It will be used by the build scripts in the rest of this chapter.

teardown.sh looks like this:

```
cat $Log
cd $Scripts
```

You do not need to modify the teardown.sh script.

Building PCSL

The next step is to build PCSL. Tell PCSL where to find the JDK with the JDK_DIR variable.

Tell PCSL the target platform and where to put output files. Set PCSL_PLATFORM to linux_i386_gcc first, then linux_arm_gcc. Tell PCSL where to put its output files with the PCSL_OUTPUT_DIR variable.

In addition, because you are cross-compiling for another platform, tell the PCSL build where to find the cross compiler by setting GNU_TOOLS_DIR.

This script, `build-pcsl.sh`, that performs the build:

```
. setup.sh

make -C $HOME/pcsl \
GNU_TOOLS_DIR=$GNU_TOOLS_DIR \
PCSL_PLATFORM=linux_arm_gcc \
PCSL_OUTPUT_DIR=$Output/pcsl \
NETWORK_MODULE=bsd/generic \
TOOLS_DIR=$HOME/tools \
TOOLS_OUTPUT_DIR=$Output/tools \
$1

if [ $? -ne 0 ]; then
  echo "build-pcsl=failed" >> $Log
else
  echo "build-pcsl=OK" >> $Log
fi

. $Scripts/teardown.sh
```

This script uses the `setup.sh` and `teardown.sh` script from before.

Run `build-pcsl.sh` as follows:

```
$ cd $HOME
/jwc
$ sh build-pcsl.sh
.
.
.
building memory port module...
make[4]: Entering directory `/jwc/pcsl/memory/memory_port'
make[4]: Leaving directory `/jwc/pcsl/memory/memory_port'
building memory module...
make[4]: Entering directory
build-pcsl=OK
$
```

A series of build messages are displayed, with a message at the end about the success of the PCSL build.

The output goes in $HOME/output/pcsl. Take a look:

```
$ cd $HOME
/jwc
$ ls ./output/pcsl
linux_arm linux_i386
$ ls ./output/pcsl/linux_arm
inc  lib  obj
$
```

Building CLDC

CLDC is built on top of PCSL. Tell CLDC to use PCSL by setting ENABLE_PCSL to true, and specify where to find PCSL with PCSL_OUTPUT_DIR.

In addition, the CLDC build system expects you to define various environment variables as follows:

- JDK_DIR is the location of the Java Development Kit software.
- JVMWorkSpace is the cldc directory.
- JVMBuildSpace contains the output of the completed CLDC build.

Define GNU_TOOLS_DIR, TOOLS_DIR, and TOOLS_OUTPUT_DIR just like for the PCSL build.

The following script, `build-cldc.sh`, runs the CLDC build.

```
. setup.sh

make -C $HOME/cldc/build/linux_arm \
GNU_TOOLS_DIR=$GNU_TOOLS_DIR \
JDK_DIR=$JDK_DIR \
ENABLE_PCSL=true \
PCSL_OUTPUT_DIR=$Output/pcsl \
JVMWorkSpace=$HOME/cldc \
JVMBuildSpace=$Output/cldc \
TOOLS_DIR=$HOME/tools \
TOOLS_OUTPUT_DIR=$Output/tools \
$1

if [ $? -ne 0 ]; then
  echo "cldc_status=failed" >> $Log
else
  echo "cldc_status=OK" >> $Log
fi

. $Scripts/teardown.sh
```

Go ahead and give it a try with `sh build-cldc.sh`. Wait for a success message at the end of the build, and take a look at the generated files in jwc/`output/cldc`.

Building Java Wireless Client Software

The last step in creating a Java Wireless Client software stack is building MIDP and optional APIs. This chapter describes how to build MIDP only. Later chapters describe how to include optional APIs in this part of the build and the many variables you can use to change how Java Wireless Client software is built.

For now, build MIDP using the CLDC and PCSL output you already generated.

You need `JDK_DIR` defined, just as it was for the CLDC build. In addition, you must tell the MIDP build where to find PCSL and CLDC:

- Set `PCSL_OUTPUT_DIR`.
- Set `CLDC_DIST_DIR` to the `dist` directory of the CLDC build you wish to use. In this case, it is `$Output/cldc/linux_arm/dist`.

The first thing that happens with the MIDP build is that tools are created for use later in the build. Use `TOOLS_DIR` to point to the `tools` directory.

Set GNU_TOOLS_DIR and TOOLS_OUTPUT_DIR as before.

Set TARGET_CPU to arm to tell the build system the type of target processor.

Finally, set MIDP_OUTPUT_DIR to the location that will contain the output of the build.

Run the build from the midp/build/linux_fb_gcc directory. Here is a script, build-sjwc.sh, that sets the necessary variables and runs the MIDP build:

```
. setup.sh

make -C $HOME/midp/build/linux_fb_gcc \
GNU_TOOLS_DIR=$GNU_TOOLS_DIR \
JDK_DIR=$JDK_DIR \
PCSL_OUTPUT_DIR=$Output/pcsl \
CLDC_DIST_DIR=$Output/cldc/linux_arm/dist \
TOOLS_DIR=$HOME/tools \
TOOLS_OUTPUT_DIR=$Output/tools \
TARGET_CPU=arm \
MIDP_OUTPUT_DIR=$Output/midp \
$1

if [ $? -ne 0 ]; then
  echo "sjwc_status=failed" >> $Log
else
  echo "sjwc_status=OK" >> $Log
fi

. $Scripts/teardown.sh
```

Run sh build-sjwc.sh without modifying the script. Wait until you see the message sjwc_status=OK at the end of the build. Congratulations! You've just built a complete MIDP stack.

Running Java Wireless Client Software

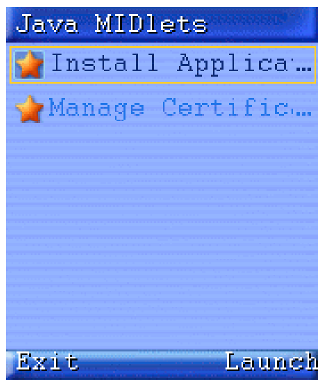
Now what do you do with it? To run the software, copy the following three directories to your Linux ARM hardware:

- $MIDP_OUTPUT_DIR/appdb
- $MIDP_OUTPUT_DIR/bin
- $MIDP_OUTPUT_DIR/lib

Now run the Java Wireless Client software with bin/arm/usertest.

The Java Wireless Client software runs on the device screen.

FIGURE 3-1 Application Manager



4

JavaCall Build System

This chapter describes the targets and options in the JavaCall build system.

JavaCall Build Overview

Building the JavaCall layer is the foundation for building other Sun Java Wireless software components. It must be built first.

The JavaCall build system has three targets that are described in the following table.

TABLE 4-1 JavaCall Build Targets

Name	Description
`all` (default)	Builds libraries and public header files in `$JAVACALL_OUTPUT_DIR`.
`clean`	Removes all output from `$JAVACALL_OUTPUT_DIR`.
`doc`	Builds API documentation in `$JAVACALL_OUTPUT_DIR/doc`.

JavaCall Build Variables

To build the JavaCall layer, you must set the same five variables for each platform. These variables are:

- `JAVACALL_DIR` - This variable points to the base JavaCall API directory, for example, `javacall.`
- `PROJECT_JAVACALL_DIR` - This variable points to the `javacall-com` directory.
- `JAVACALL_OUTPUT_DIR` - This variable tells the build system where to put the output of the JavaCall API build.

In addition to setting the three variables above, building the JavaCall layer requires setting two additional variables:

- TOOLS_DIR - This variable points to the location of the build tools used to build JavaCall.
- TOOLS_OUTPUT_DIR - This variable points to the location of tools-specific output.

Extending the Build System

If you wish to extend the JavaCall build system to other platforms not currently supported by the Sun Java Wireless Client software, here is what you need to know.

JavaCall Directory Structure

There are two JavaCall build directories in the Sun Java Wireless Client software, javacall and javacall-com. Each contains a different implementation of the JavaCall interfaces, makefiles, and build scripts. However, the directory structure of each is similar.

The JavaCall directory javacall-com contains the following four subdirectories:

- build - This directory contains compiler-specific makefiles, as well as other commonly shared makefiles. The supported compilers are Microsoft Visual C++ (vc) for the win32_emul platform and Gnu Compiler Collection (gcc) for the Linux_ARM platform.

  To support a new compiler, you must create a corresponding folder, then copy and modify the makefiles provided here for the new compiler.
- interface - this directory is common for all platforms and contains the JavaCall API header files.
- implementation - This directory contains the JavaCall API for a specific implementation. For example, for the Sun Java Wireless Client software, this directory contains the JavaCall APIs for the win32_emul platform. It also contains "stubbed" implementation files for Linux.

  To port JavaCall to a new platform, the stubbed implementation must be copied to an appropriate folder (for example, implementation/linux_arm). Then the stubbed functions must be filled in with code for your implementation.
- configuration - This directory contains makefiles created for a specific project. For example, you may have only one platform, such as win32_emul, but have two or three projects based on that platform. The makefiles for the project configurations go into this directory.

5

PCSL Build System

This chapter describes the targets and options in the PCSL build system. It also includes an overview of the process for performing a PCSL build for your own platform.

PCSL Build Overview

You must define PCSL_PLATFORM to build PCSL. The value of this variable specifies the software layer below PCSL, the target processor, and the compiler type. For example, linux_arm_gcc specifies that PCSL is implemented on top of Linux on an ARM processor, and the build is performed using GCC. Another example is javacall_i386_vc, which means that PCSL is implemented over the JavaCall API on Intel x86 hardware, and Microsoft's Visual C++ compiler is used for the build.

You can find the supported values for PCSL_PLATFORM by looking in the pcsl/makefiles/platforms directory.

The PCSL build system includes four targets which are described in the following table.

TABLE 5-1 PCSL Build Targets

Name	Description
all (default)	Builds libraries and public header files in $PCSL_OUTPUT_DIR.
clean	Removes all output for $PCSL_PLATFORM from $PCSL_OUTPUT_DIR.
doc	Builds API documentation in $PCSL_OUTPUT_DIR/doc.
donuts	Builds unit tests.

The name all is a little misleading, as it does not build the documentation or the unit tests. However, all does build the libraries and headers that are needed to build CLDC and the Java Wireless Client software.

Output

The output of the PCSL build is placed in PCSL_OUTPUT_DIR. If you do not define this variable, the default value is pcsl/output. The PCSL build system creates one output subdirectory per platform (minus the compiler). For example, the output of a javacall_i386_vc build goes in $PCSL_OUTPUT_DIR/javacall_i386.

Building all creates the following directories in the platform output directory:

- inc
- lib
- obj

If you also build donuts, you'll get two additional directories:

- bin
- generated

CLDC and the Java Wireless Client software use the inc and lib directories.

Debugging Symbols

PCSL is built without debugging symbols by default. Set USE_DEBUG to true to include debugging symbols in the output of the build.

API Documentation

The PCSL build system uses Doxygen to create API documentation for the doc target. Doxygen is available at:

http://doxygen.org/

The PCSL build system assumes that Doxygen is located at /usr/bin/doxygen. If this is not true, edit pcsl/makefiles/share/Docs.gmk and change the definition of the DOXYGEN_CMD variable. For example, on Windows, you might have to do something like this:

```
DOXYGEN_CMD = c:/PROGRA~1/doxygen/bin/doxygen.exe
```

When the `doc` build finishes, the top page of the documentation is `$PCSL_OUTPUT_DIR/doc/doxygen/html/index.html`.

Selecting Modules

PCSL includes five services, as follows:

- `file`
- `memory`
- `network`
- `print`
- `string`

The implementation of a service is a *module*. When you build PCSL, you can select which module to use for each service. A variable is used to select each service's module.

Each PCSL service has a corresponding directory. In general, modules are represented by the subdirectories of each service directory. For example, the `print/file` directory contains the `file` module of the `print` service.

To build the `file` module for the `print` service do something like this:

```
$ make PRINT_MODULE=file
```

The following table shows the variable names and possible values for module selection.

TABLE 5-2 PCSL Module Selection

Service Variable	Values
FILE_MODULE	armsd javacall posix (default) ram simulates a file system in RAM stubs win32
MEMORY_MODULE	heap dynamically allocates memory within a chunk of heap. This module has an additional debugging facility to help detect memory leaks and corruption. malloc (default) uses standard C malloc() stubs
NETWORK_MODULE	bsd/generic uses the BSD implementation. javacall sos is Server over Serial stubs socket/winsock
PRINT_MODULE	file prints to a log file javacall stdout (default) stubs
STRING_MODULE	utf8 utf16 (default)

About Stubs

Each PCSL service includes a stubs module that builds empty versions of its public API. The empty versions of the functions that have a return value return either NULL or an appropriate error code.

The stubs modules make it easy to port the Java Wireless Client software to a new platform, one PCSL service at a time. You can build the Java Wireless Client software before you port all of the services by using stubs for any services that are not yet ported.

Building Individual Services

Each of the build targets (all, clean, doc, and donuts) that you can use at the top level of PCSL can also be applied to builds in the directories for each service. Simply change your working directory to one of the service directories, set variables, and run make.

For example, to build just the print service using the file module, do something like this:

```
D:\jwc>cd pcsl\print

D:\jwc\pcsl\print>make PRINT_MODULE=file
```

Network Service

Datagram and server socket support in the network service is controlled by the USE_DATAGRAM and USE_SERVER_SOCKET variables. The default value for both is true. To disable datagrams or server sockets, set the corresponding variable to false.

Note – The serial subdirectory of the network service is not a module. The files in the serial subdirectory are used by the sos module.

If you choose the Server over Serial (sos) module, the build is slightly more complicated. The sos module is based on the Java Communications API. you will need a Java Communications API implementation for your platform, and you will need a Java platform compiler.

First, set JDK_DIR to point to your JDK software. Next, install the Java Communications API implementation for your operating system. Implementations for Linux and the Solaris™ Operating System are available at:

http://java.sun.com/products/javacomm/.

Follow the Downloads link. Implementations for other platforms are available at:

http://www.rxtx.org/

Once the JDK software and Java Communications API are available, you can build the sos module.

Running Unit Tests

Building the unit tests creates the executable for running the tests. The executable is `bin/donuts` in the output directory for your platform. Use it to run all the tests or an individual test and to list the unit tests.

To run all the tests, just run `donuts`. This example shows how to run unit tests for the `linux_i386` platform:

```
$ cd $PCSL_OUTPUT_DIR/linux_i386
```

```
$ bin/donuts
```

List the unit tests like this:

```
$ bin/donuts -list
```

Run a test by naming it. This example runs `testString`:

```
$ bin/donuts testString
```

Extending the Build System

The PCSL build system uses a top-level platform-specific makefile in the `pcsl/makefiles/platforms` directory. It also uses makefiles for specific operating systems and CPUs. These makefiles are in `pcsl/makefiles/share`.

The makefiles, which have `.gmk` extensions, define all of the variables exported to the subsystem makefiles. The variables usually specify directories. The makefiles also define OS-specific commands, such as `CC_OUTPUT`, and subsystem targets.

This section outlines the steps for extending the build system to support a new platform, or extending the build system for a new service module.

You can find more information in `pcsl/makefiles/README.build_port`.

Creating a New Platform Makefile

First, create a platform makefile in `pcsl/makefiles/platforms`. Use an existing makefile as a template. Choose one that is close to your platform and compiler. Name the new file using the file naming convention of OS, processor, and compiler names, separated by underscores.

For example, if you want to build for Symbian OS, using the ARM processor, with the GCC compiler, copy the `linux_arm_gcc.gmk` file. Name your new file `symbian_arm_gcc.gmk`.

Next, define `PCSL_OS`, `PCSL_CPU`, and other platform-specific variables in your new makefile.

For example, if you are configuring the build system for the Linux OS and are using the ARM CPU, set the following variables:

- `PCSL_OS=linux`
- `PCSL_CPU=arm`

If you wish to override the `PCSL_CHUNKMEM_DIR` and `PCSL_CHUNKMEM_IMPL` values that are defined in `pcsl/makefiles/top.gmk`, define them in your platform makefile.

Include the matching OS and compiler makefiles from `pcsl/makefiles/share` in your platform makefile. The next section describes how to create OS and compiler makefiles if they are not available.

Creating New OS and Compiler Makefiles

If your platform includes an OS which does not have a corresponding makefile in `pcsl/makefiles/share`, you must create a new one. Use an existing makefile as a template. Give the new file your operating system name.

For example, if you are configuring the build system for the Symbian OS, use `linux.gmk` as a template to create `symbian.gmk`.

The OS makefile defines operating-system specific variables. For example, `EXE` is the suffix for executables.

Likewise, if your platform includes a compiler that does not have a corresponding makefile in `pcsl/makefiles/share`, create a new one. Use an existing makefile as a template. Give the new file your compiler name.

Define the compiler commands and flags in the compiler makefile. The following variables must be defined:

- `CC` - Compilation command for C files (such as `gcc`)
- `CPP` - Compilation command for C++ files (such as `g++`)
- `LD` - Link command for C++ and C files (such as `g++`)
- `AR` - Archiving command (such as `ar -rc`)
- `CC_OUTPUT` - Output flag for C and C++ files (such a, `-o`)
- `AR_OUTPUT` - Output flag for the archiver (such as `/OUT:`)

- LD_OUTPUT - Output flag for the linker (such as -o)
- LIB_EXT - Suffix for library files (such as .a)
- CFLAGS - Compiler flags (such as -c -O3)
- LD_FLAGS - Linker flags, if any

If your target platforms have the same OS on all devices, but different CPUs, add compiler flags based on the CPU of your target platform.

Creating a New Module

To create a new module for a service, first create a subdirectory in the service directory. For example, if you create a new platform implementation, *my-platform*, of the file service, create its implementation directory, file/*my-platform*.

Create a file named GNUmakefile file in the new directory. Use an existing GNUmakefile from another module directory as a template. For example, in the file service module, the posix and ram directories have GNUmakefile files you can use as a model

Keep these points in mind when you use an existing GNUmakefile as a template:

- You do not need to change the donuts and doc target definitions and rules.
- You might need to change the file names in the all and clean target definitions.
- You might need to change the compilation rules for the files, but existing rules work in many cases.

6

CLDC Build System

This chapter provides a very brief description of building CLDC. It also includes a section that describes build variables that must match between the CLDC HotSpot Implementation build and the Java Wireless Client software build.

CLDC HotSpot Implementation has its own set of documentation. For detailed information about the build system, read the *CLDC HotSpot Implementation Porting Guide*.

CLDC Build Overview

The CLDC HotSpot Implementation build system expects you to define three variables:

- `JDK_DIR` is the location of the Java Development Kit software.
- `JVMWorkSpace` is the `cldc` directory.
- `JVMBuildSpace` is the location of the CLDC HotSpot Implementation build output.

If you build CLDC HotSpot Implementation on top of PCSL, set `ENABLE_PCSL` to `true`, and specify where to find PCSL with `PCSL_OUTPUT_DIR`.

Likewise, if you build CLDC HotSpot Implementation on top of the JavaCall API, tell the build system where to find the JavaCall API with `JAVACALL_OUTPUT_DIR`.

To perform the build, find the `cldc/build` subdirectory which corresponds to the platform for which you are building. For example, if you want to build the Java Wireless Client software for Windows, perform the CLDC HotSpot Implementation build in `cldc/build/win32_i386`.

CLDC Build System Variables

TABLE 6-1 lists the variables used to build the CLDC HotSpot Implementation.

TABLE 6-1 CLDC Build System Variables

CLDC HotSpot Implementation	Description
ENABLE_JAVA_DEBUGGER	Provides KDWP support.
ENABLE_WTK_PROFILER	Provides support for profiling the Java platform.
ENABLE_MONET	Provides fast loading class format.
ENABLE_CLDC_11	Provides support for the CLDC 1.1 Specification.
ENABLE_ISOLATES	Provides multitasking functionality.
ENABLE_VERIFY_ONLY	Provides improved MIDlet startup time by preverifying a MIDlet's classes when the MIDlet is installed instead of every time it runs.

Note – Building the MIDP component of the Sun Java Wireless Client requires you to set build variables that correspond to the variables used to build CLDC. Make note of the variables you set for CLDC. The corresponding variables needed for MIDP are described in detail in Chapter 7.

7

Java Wireless Client Software Build System

The build system for the Java Wireless Client software builds MIDP and any optional APIs you wish to include. This build system is based in the `midp` directory, but it is capable of including optional API source code from other locations.

Overview

To perform the build, find the `midp/build` subdirectory that corresponds to the platform for which you are building. For example, if you want to build the Java Wireless Client software on top of the JavaCall API, perform the build in `midp/build/javacall`.

You need to tell the build system about the lower layers upon which you are building, as follows:

- Set `CLDC_DIST_DIR` to the `dist` directory of the CLDC build.
- Set `PCSL_OUTPUT_DIR` to the output directory of the PCSL build.
- If you are using the JavaCall API, set `JAVACALL_OUTPUT_DIR` to the same value you used when you built the JavaCall API. Also set `JAVACALL_PLATFORM` to the right value for your target platform.

You must also set `JDK_DIR` to point to your installation of the JDK software.

In addition, the Java Wireless Client software build uses some internal tools. Use `TOOLS_DIR` to point to the `tools` directory.

Finally, the output of the Java Wireless Client software build is placed in `MIDP_OUTPUT_DIR`.

TABLE 7-1 lists and describes the build targets for the Java Wireless Client software build system.

TABLE 7-1 Java Wireless Client Software Build Targets

Target	Description
`all` (default)	Creates the executables, classes, and tools for an implementation of the Java Wireless Client software.
`parallel_all`	Makes possible more effective building of Java Wireless Client modules, by allowing more than one make to be run in parallel. Example: `$ make parallel_all NUM_JOBS=4`
`docs_html`	Generates Porting API documentation. Define `DOXYGEN_CMD` to point to your installation of Doxygen.
`clean`	Removes all build output files.

Output

Many output directories are placed in `$MIDP_OUTPUT_DIR`, but you do not need all of it. The `appdb`, `bin`, and `lib` directories contains everything you need to run an executable on your target device.

Debugging Symbols

TABLE 7-2 shows the build options that control the build's debug configuration. All of the options are `false` by default. To enable a debugging option, set it to `true`.

TABLE 7-2 Debugging Selection Options

Name	Description
USE_DEBUG	Builds an implementation that enables debug in the build. This preserves symbols in both class file and executables. Useful when Java platform stack trace is needed or native debugger is used.
USE_I3_TEST	Build an implementation with unit tests. Run all unit tests with `bin/i3test` command.
USE_JAVA_DEBUGGER	Enables Java platform debugger support, also known as KDWP. Allows debugging on installed MIDlet classes. To enable debugging on ROMized system classes, use this option with `USE_DEBUG=true`. Example: `USE_JAVA_DEBUGGER=true`
USE_JAVA_PROFILER	Enables CLDC HotSpot Implementation profiler feature, used by the Sun Java Wireless Toolkit.

When you want to make adjustments to the debugging options, first `make clean` to get a fresh start. Then set debugging options and build the Java Wireless Client software again.

API Documentation

The Java Wireless Client software generates API reference documentation using Doxygen and `javadoc`. Doxygen is available at:

`http://doxygen.org/`

The build system assumes that Doxygen is located at `/usr/bin/doxygen`. You can override this value on the command line by defining `DOXYGEN_CMD`.

`javadoc` is part of the JDK software, so it is found as long as `JDK_DIR` is defined.

Build Options

This section describes options you can use to control the Java Wireless Client software build.

CLDC Selection

TABLE 7-3 shows the build options that control the CLDC technology in the build.

TABLE 7-3 CLDC Selection Options

Name	Values	Description
USE_CLDC_11	true (default) false	Builds an implementation that is compliant with the CLDC 1.1 Specification. If this variable is false, CLDC 1.0 is used instead.
USE_CLDC_RELEASE	false (default) true	For non-debug builds, setting this variable to false means that the product version of CLDC is linked. This provides the best footprint and performance, with no debug or tracing facilities.
USE_MONET	false (default) true	Enables on-device support (conversion and loading) of binary application image files for fast class loading.

Mapping Configuration Variables

Some CLDC HotSpot Implementation and Sun Java Wireless Client software features must match. When building MIDP, you must set variables and values that correspond to those used when building the CLDC HotSpot Implementation described in Chapter 6.

In general, the CLDC Hotspot Implementation build options have the prefix ENABLE_, and the Java Wireless Client software build options have the prefix USE_. TABLE 7-4 lists build options in the two build systems.

TABLE 7-4 Configuration Options Mapping Between Build Systems

CLDC HotSpot Implementation	SJWC Software Client	Description
ENABLE_JAVA_DEBUGGER	USE_JAVA_DEBUGGER	Provides KDWP support.
ENABLE_WTK_PROFILER	USE_JAVA_PROFILER	Provides support for profiling the Java platform.
ENABLE_MONET	USE_MONET	Provides fast loading class format.
ENABLE_CLDC_11	USE_CLDC_11	Provides support for the CLDC 1.1 Specification.
ENABLE_ISOLATES	USE_MULTIPLE_ISOLATES	Provides multitasking functionality.
ENABLE_VERIFY_ONLY	USE_VERIFY_ONCE	Provides improved MIDlet startup time by preverifying a MIDlet's classes when the MIDlet is installed instead of every time it runs.

Module Selection

TABLE 7-5 shows the build options that control which LCDUI and AMS implementations are in the build.

TABLE 7-5 Module Selection Options

Name	Value	Description
SUBSYSTEM_LCDUI_MODULES	chameleon (default)	chameleon builds an implementation that uses adaptive user-interface technology.
USE_NATIVE_AMS	false (default) true	When false, builds an implementation that uses the AMS written in the Java programming language. When true, builds an implementation that uses a native AMS.

Native AMS Image Resource Policy

Image resources in a native AMS are usually stored in PNG format. However, decoding compressed PNG images takes time while the Java Wireless Client software starts up. To start faster, images can be stored in a platform-dependent raw format that is faster to load.

When you set the build option `USE_NATIVE_AMS=true`, you can improve startup performance by also setting `USE_RAW_AMS_IMAGES=true`. The trade-off for this optimization is an increase in memory needed to store the raw image resources.

If the build option `USE_NATIVE_AMS=true`, and the build option `USE_RAW_AMS_IMAGES=false`, the original PNG image resources of the AMS are used during the build process.

Multitasking

`USE_MULTIPLE_ISOLATES` controls multitasking in the build. When set to `true`, the Java Wireless Client software can run more than one MIDlet at the same time. The default is `false`, which means that only one MIDlet can be run at a time.

Startup Performance

The `USE_VERIFY_ONCE` option determines whether a MIDlet's classes are preverified when the MIDlet is installed (`true`) or every time the MIDlet is run (`false`). Preverifying during installation (`true`) improves MIDlet startup time.

Resource Allocation Policy

When `USE_FIXED` is `false`, the implementation uses the default resource policy, open-for-competition. This is the default value.

When `USE_FIXED` is `true`, the implementation uses a fixed partition resource policy.

Cryptography Selection

TABLE 7-6 shows the build options that control the cryptography configuration in the build.

TABLE 7-6 SSL Selection Options

Name	Values	Description
USE_SSL	true false (default)	Controls whether the implementation uses SSL. This does not affect security for OTA and SATSA. For that, see USE_RESTRICTED_CRYPTO.
USE_RESTRICTED_CRYPTO	true false (default)	Includes ciphers and the features that depend on ciphers, which are secure OTA, SecureConnection, HTTPS, and SATSA-CRYPTO (JSR 177).
RESTRICTED_CRYPTO_DIR		Required when USE_RESTRICTED_CRYPTO=true. Customers must provide their own crypto library.

Server Socket Selection

USE_SERVER_SOCKET controls whether server socket support is included in the build. Use true, the default value, to include server socket support.

Runtime Java Platform Properties Selection

Java platform properties can be hard-coded in the Java Wireless Client software, or they can be loaded from files at runtime. Hard-coding is faster, but using files is more flexible.

USE_STATIC_PROPERTIES controls this behavior. The default value is true, which means hard-coded values are used. Set USE_STATIC_PROPERTIES to false to load properties from files. The default file locations are lib/internal.config and lib/system.config in MIDP_OUTPUT_DIR.

Specifying a Target CPU and Device

The TARGET_CPU option specifies the type of processor for which you are building the software. TARGET_DEVICE makes it possible to be even more specific.

Use the TARGET_DEVICE build option to select configuration parameters within a specific platform. Valid values are omap730, zaurus, and x86. This value is appended to the names of input files for the configurator tool (see the *Porting Guide* for details).

For linux_fb builds, the default value of TARGET_DEVICE is x86. However, if TARGET_CPU is set to arm, then TARGET_DEVICE will default to omap730.

For javacall and win32 builds, the default value for TARGET_DEVICE is x86.

Build Constraints

Some build options can be used together and others cannot. For example, some build combinations require the option USE_MULTIPLE_ISOLATES=true for other options to work.

Following are the known constraints when building the Java Wireless Client software:

- If USE_NATIVE_AMS=true, you must also set USE_MULTIPLES_ISOLATES=true.
- If USE_FIXED=true, you must also set USE_MULTIPLES_ISOLATES=true.
- If USE_MONET=true, you must set USE_VERIFY_ONCE=false, and vice-versa.
- If USE_SSL=true, you must also set USE_RESTRICTED_CRYPTO=true.

Building Optional Package APIs

The Java Wireless Client software supports the following JSRs and their corresponding build flags.

- JSR 75 - File Connection and Personal Information Management API
- JSR 82 - Bluetooth API
- JSR 120 - Wireless Messaging 1.0
- JSR 135 - Mobile Media API
- JSR 172 - Java Web Services API
- JSR 177- Security and Trust Services API
- JSR 179 - Location API
- JSR 180 - Session Initiation Protocol API
- JSR 205 - Wireless Messaging 2.0
- JSR 211 - Content Handler API
- JSR 226 - Scalable 2D Vector Graphics API
- JSR 229 - Payment API
- JSR 234 - Advanced Multimedia Supplements API
- JSR 238 - Mobile Internationalization API
- JSR 239 - Java Binding for the OpenGL™ Embedded Subset API
- JSR 256 - Mobile Sensor API
- JSR 280 - XML API for Java ME

Some JSRs have complicated builds. The section "Optional Package API Details" on page 51 describes how to build these JSRs.

About Optional JSRs

Most JSRs define a single optional package API, but some contain more than one. For example, JSR 75 includes a Personal Information File Management (PIM) API and a File Connection API. JSR 177 defines four distinct optional APIs.

The build flags described in this section affect the full set of APIs defined in a JSR. For example, if you want to include the JSR 75 PIM and File Connection APIs, set `USE_JSR_75` to `true` and set `JSR_75_DIR` to `$HOME/jsr75`, where `$HOME` is the installation directory of the Java Wireless Client software. Both components of the API will be built.

Optional Package JSRs and Other Build Components

To build a specific JSR in the MIDP (client), you must also build that JSR in other components, too. For example, to build Sun Java Wireless Client software with the Wireless Messaging API (JSR 120), you must set `USE_JSR_120=true` when you build both the JavaCall and MIDP components.

Note – When building the Sun Java Wireless Client software, you must set the same variables the same way when building both the MIDP and JavaCall components. For example, if `USE_JSR_XXX=true` in the MIDP component build, it must also be `true` when building JavaCall. However, this rule does not apply just to optional package JSRs. It applies to the other variables discussed in this chapter, too.

Optional API Variable Pairs

In general, optional package APIs can be included by defining two pairs of variables. The first pair, the JSR pair, tells the build system to use a specific optional package API (`USE_JSR_XXX`) and then tells the build system where to find the source code for that API (`JSR_XXX_DIR`).

The second pair, the Abstractions pair, tells the build system to use the Abstractions component (`USE_ABSTRACTIONS`) and where to find the source code for the Abstractions component (`ABSTRACTIONS_DIR`). The abstractions component provides the interfaces that allow optional package JSRs to make calls to MIDP and the virtual machine.

The `ABSTRACTIONS_DIR` variable must point to the location of the abstractions component, for example,
`ABSTRACTIONS_DIR=$COMPONENTS_DIR/abstractions`.

Note – In the Sun Java Wireless Client software build scripts described in Chapter 2, the `$COMPONENTS_DIR` variable is set to `$HOME`, the location of your home directory. However, this may differ in your makefiles and build scripts. Make sure when you set `$ABSTRACTIONS_DIR` that the resolved path points to the correct location of your `abstractions` directory.

Using the JSR Variable Pair

There are two ways JSR variable pairs can be used to build the Sun Java Wireless Client software:

- By adding them as command strings to the SJWC build makefiles
- By using them on the command line, as subcommands to the make utility

Optional package JSR build variables can be added to the SJWC makefiles, following standard makefile conventions. They can also be added to the SJWC build scripts provided in Chapter 2. It depends on how you want your build system to proceed.

In the case of JSR variable pairs, the following guidelines must be followed:

- All USE_JSR_XXX variables *must* be defined on the make command line.
- JSR_XXX_DIR variables can be defined in the makefiles, the build scripts, or on the make command line, as you choose.

Note – The USE_JSR_XXX variable is set to false in all SJWC makefiles for all optional package JSRs. If you add an optional package JSR to your build, to ensure the false value is overridden, you *must* set USE_JSR_XXX=true on the make command line for each SJWC component you build.

For more information on the make command line, see "Using the make Command Line" on page 50.

Using the ABSTRACTIONS Variable Pair

The USE_ABSTRACTIONS=true variable is used in tandem with the JSR variable pairs. That is, if any of the following optional package JSRs is added to the build by setting USE_JSR_XXX=true, then USE_ABSTRACTIONS=true must also be set:

- JSR 75
- JSR 82
- JSR 120
- JSR 135
- JSR 172
- JSR 177
- JSR 179
- JSR 180
- JSR 205
- JSR 211
- JSR 226
- JSR 229
- JSR 234
- JSR 238

Note – Unlike USE_JSR_XXX, which must be individually set for each JSR added to the build, USE_ABSTRACTIONS is only set once. If any of the JSRs listed above is added to the build, you must set USE_ABSTRACTIONS=true. However, once it's set, it does not need to be set again for any additional JSRs.

In the case of Abstractions variable pairs, the following guidelines must be followed:

- All USE_ABSTRACTIONS variables *must* be defined on the make command line.
- ABSTRACTIONS_DIR variables can be defined in the makefiles, the build scripts, or on the make command line, as you choose.

Note – The USE_ABSTRACTIONS variable is set to false in all SJWC makefiles. To ensure the false value is overridden, you *must* set USE_ABSTRACTIONS=true on the make command line for each SJWC component you build.

For more information on the make command line, see "Using the make Command Line" on page 50.

Building JSRs Without the USE_ABSTRACTIONS Variable

The following optional package JSRs can be built without the USE_ABSTRACTIONS variable (that is, with this variable set to false):

- JSR 239
- JSR 256
- JSR 280

Using the make Command Line

To define build variables in the make command line, use the following generic command:

D:\>make *subcommand1 subcommand2 ... target*

In the make command line above, *subcommand* is the variable you want to set.

For example, to define the variable USE_JSR_120 on the make command line, the command string would look like the following:

D:\> make USE_JSR_120=true *target*

In most cases, the *target* is all, which means "build everything," as described in TABLE 7-1.

Note – You can define any SJWC build variable on the make command line, not just optional package JSR variables.

Using the make Command Line with Makefiles

Variables defined as subcommands on a make command line override any setting for that variable in the build makefiles. For example, if USE_ABSTRACTIONS=true is set on the make command line, it overrides any false setting for it in the makefiles. However, a variable set this way is good for only one build. It does not reset the value of the variable in the makefiles themselves.

To set the variable again (without altering the value in the makefile), you must reenter the variable on the command line each time the make command is run.

Optional Package API Details

This section describes additional steps that need to be performed in order to build some of the optional package APIs.

Building JSR 120 and JSR 205

Unlike most JSRs, which can be built independently of each other, a dependency exists between the Wireless Messaging 1.0 API (JSR 120) and the Wireless Messaging 2.0 API (JSR 205). JSR 205 is a superset of JSR 120.

To properly build JSR 205, you must have an implementation of JSR 120 available, and USE_JSR_120 and USE_JSR_205 must both be set to true.

You must also set JSR_120_DIR and JSR_205_DIR to the location of your JSR 120 and JSR 205 source files, for example:

```
JSR_120_DIR=$HOME/jsr120
JSR_205_DIR=$HOME/jsr205
```

Note – To build JSRs 120 and 205, you must set USE_ABSTRACTIONS=true. For more information, see "Using the ABSTRACTIONS Variable Pair" on page 49.

Building JSR 135 and JSR 234

Like the relationship between JSR 120 and JSR 205, a dependency exists between the Mobile Media API (JSR 135) and the Advanced Multimedia Supplements API (JSR 234). JSR 234 is a superset of JSR 135.

To properly build JSR 234, you must have an implementation of JSR 135 available, and `USE_JSR_135` and `USE_JSR_234` must both be set to `true`.

You must also set `JSR_135_DIR` and `JSR_234_DIR` to the locations of your JSR 135 and JSR 234 source files, for example:

```
JSR_135_DIR=$HOME/jsr135
JSR_234_DIR=$HOME/jsr234
```

Note – To build JSRs 135 and 234, you must set `USE_ABSTRACTIONS=true`. For more information, see "Using the `ABSTRACTIONS` Variable Pair" on page 49.

The `USE_JPEG` Variable

Support for JPEG image format is a requirement of the MSA-248 Specification and used in building MIDP. Therefore, this variable should always be `USE_JPEG=true`.

`USE_JPEG=true` is also required to build the JSR 234 optional package.

Building JSR 226 and JSR 172

The JSR 226 Scalable Vector Graphics (SVG) API uses a renderer called Pisces. When you set `USE_JSR_226` to `true`, you must also set `PISCES_DIR` to point to the Pisces source files. For example:

```
USE_JSR_226=true
PISCES_DIR=$HOME/pisces
```

JSR 226 uses the XML parser that is part of JSR 172 (Web Services). If you plan to include JSR 226 in your build, you must also include JSR 172, as shown here:

```
USE_JSR_172=true
JSR_172_Dir=$HOME/jsr172
```

Note – To build JSR 172 and JSR 226, you must set `USE_ABSTRACTIONS=true`. For more information, see "Using the `ABSTRACTIONS` Variable Pair" on page 49.

Building JSR 229 and JSR 120

Like other optional package JSRs mentioned in this chapter, a dependency exists between the Payment API (JSR 229) and the Wireless Messaging 1.0 API (JSR 120).

To properly build JSR 229, you must have an implementation of JSR 120 available, and `USE_JSR_120` and `USE_JSR_229` must both be set to `true`.

You must also set `JSR_120_DIR` and `JSR_229_DIR` to the locations of your JSR 120 and JSR 229 source files, for example:

```
JSR_120_DIR=$HOME/jsr120
JSR_229_DIR=$HOME/jsr229
```

Note – To build JSRs 229 and 120, you must set `USE_ABSTRACTIONS=true`. For more information, see "Using the `ABSTRACTIONS` Variable Pair" on page 49.

Building JSR 177

To properly build the Security and Trust Services APIs (JSR 177) optional package, you must set the `USE_JSR_177` and `JSR_177_DIR` variables as described previously. You must also have the Java Card™ Development Kit 2.2.1 installed on your build platform.

The Java Card Development Kit is available at:

`http://java.sun.com/products/javacard/dev_kit.html`.

Set `JC_DIR` to the location of your Java Card Development Kit 2.2.1 installation, like this:

```
JC_DIR=/java_card_kit-2.2.1
```

Note – To build JSR 177, you must set `USE_ABSTRACTIONS=true`. For more information, see "Using the `ABSTRACTIONS` Variable Pair" on page 49.

JSR 177 has additional build options for the SATSA -APDU package, as described in TABLE 7-7.

TABLE 7-7 APDU Build Options

Option	Description
`JSR_177_APDU_MANAGER`	Selects the CardDevice implementation of the APDU package. Possible values are `carddevice`.
`JSR_177_APDU_CARDDEVICE`	Possible values are `platformcarddevice` or <*yourcarddevice*>. This option is required only if the value of `JSR_177_APDU_MANAGER` is `carddevice`.
`JSR_177_USE_EMULATOR`	Specifies if the SAT applications for testing emulator are generated. Possible values are `true` or `false`.

Building JSR 239

JSR 239 defines a Java platform API binding for the industry standard OpenGL® ES API. Your job is to bind the JSR 239 API to the underlying OpenGL ES API for your device.

This binding process is the same regardless of whether you are using the JavaCall API or not. The implementation of JSR 239 in Java Wireless Client software bypasses the JavaCall layer and binds directly to your device's OpenGL ES API.

In most cases, you will be binding to a third-party OpenGL ES engine. Then the only task is to tune the build system to collect together Java Wireless Client software and the third-party engine.

Use the following as your reference makefile and change it according to the OpenGL ES implementation you are using:

```
jsr239/src/cldc-oi/config/sample.gmk
```

You must also set the following build variables:

```
USE_JSR_239=true
JSR_239_DIR=$HOME/jsr239
```

Note – Your path to the OpenGL ES engine source code may be different than shown here. Set it according to your implementation

---

Working With Stubs

The Java Wireless Client software build system uses a top-level platform-specific makefile in `midp/build`, and OS-specific and compiler-specific makefiles in `midp/build/common/makefiles`.

The makefiles define all the variables (usually specifying directories) that are exported to the subsystem makefiles. They also define OS specific commands and subsystem targets.

This section shows you how to configure the Java Wireless Client software build system for your new platform. Because porting to a new platform begins with successfully building *stubs* (empty or generic platform-independent versions of functions), this section begins by showing you how to configure for a `stubs` build.

After you can build the `stubs` implementation for your platform, incrementally fill the stubs with working code and rebuild. Use the instructions at the end of this section to further modify the build system, if necessary, through each build cycle.

Configuring the Build System for Stubs

First, create a directory for your platform in `midp/build`.

Use the naming convention of operating system, CPU, and compiler, separated by underscores, for the directory. For example, if the OS is Symbian, the CPU is ARM, and the compiler is GCC, create the following directory:

```
midp/build/symbian_arm_gcc
```

Next, copy `GNUmakefile` and `Options.gmk` from `linux_stubs_gcc` into your new directory.

Modify definitions in your new `GNUmakefile` for your platform. `TARGET_PLATFORM` should stay as `stubs`. Modify the values in `Options.gmk` for your platform.

If you need platform-specific definitions, create `stubs.gmk` in your new platform directory. Platform-specific definitions might mean extra include files, extra flags for your compiler, and so on.

Make sure to include `stubs.gmk` in your `GNUmakefile`.

Create *compiler*`-jtwi.gmk`, *compiler*`.gmk`, and *os*`.gmk` in `midp/build/common/makefiles`.

The file *compiler*-jtwi.gmk contains MIDP-specific directory and library definitions. See gcc-jtwi.gmk in midp/build/common/makefiles for comments and a list of definitions. Consider copying, renaming, and editing gcc-jtwi.gmk.

The file *compiler*.gmk contains generic compiler definitions. See gcc.gmk in the midp/build/common/makefiles for a list of required compiler definitions. Consider coping, renaming, and editing gcc.gmk.

The file *os*.gmk contains generic OS-specific definitions. See linux.gmk in the midp/build/common/makefiles for a list of required definitions. Consider coping, renaming, and editing linux.gmk.

Note – *compiler* and *os* must match the HOST_COMPILER and HOST_OS values in GNUmakefile.

You are now ready to launch the stubs build.

Updating the Build System for Filled-in Stub Functions

After you successfully build the stubs implementation, incrementally implement the empty platform-specific functions. After implementing a platform-specific function, rebuild stubs and, if required, modify the build system.

If you add a new native file as you implement a stub function, you must modify the corresponding subsystem's .cfg file. A .cfg file has definitions such as the source file path, extra include files, and a list of native files. Each subsystem has a stubs.cfg in its config directory. For example, the core subsystem has a stubs.cfg file in src/core/config.

Updating the Source Files and Build System after Porting

After you complete all of the stub functions, meaning you can build the entire system for your platform, consider renaming the stubs to match the name of your platform. It is not a functional change, but is a good way to indicate to yourself and any team members that the milestone of a full platform-specific build has been reached.

To rename stubs, change the TARGET_PLATFORM variable in the file GNUmakefile to the name of your platform. Then rename all directories named stubs to the name of your platform.

Glossary

API Application Programming Interface. A set of classes used by programmers to write applications, which provide standard methods and interfaces and eliminate the need for programmers to reinvent commonly used code.

AMS Application Management Service. The system functionality that completes tasks such as installing applications, updating applications, and switching foregrounds.

Application list The screen that lists all of the installed applications. The user gets to this screen by pressing the `Apps` soft key on the home screen. The application list uses text color to show which applications are running. It also provides a system menu that enables the user to perform application management tasks on the highlighted application.

Background An application state in which the application does not receive events from its input stream and its displayable is not rendered to the screen.

CDC Connected Device Configuration. A Java ME platform configuration for devices, it requires a minimum of 2 megabytes of memory and a network connection that is always on.

CLDC Connected Limited Device Configuration. A Java ME platform configuration for devices with less than 512 kilobytes of RAM and an intermittent (limited) network connection, it uses a stripped-down Java virtual machine called the KVM, as well as several minimalist Java platform APIs for application services.

Configuration Defines the minimum Java runtime environment (for example, the combination of a Java virtual machine and a core set of Java platform APIs) for a family of Java ME platform devices.

Foreground The application state in which the application is rendered to the device display and the input stream is passed to it.

Foreground switching Changing which application is in the foreground by shifting the focus from one application to another.

GCF Generic Connection Framework. A part of CLDC, it improves network connectivity for wireless devices.

Home screen The main screen of the application manager. This is the screen the user sees after they exit an application.

HTTP HyperText Transfer Protocol. The most commonly used Internet protocol, based on TCP/IP, which is used to fetch documents and other hypertext objects from remote hosts.

HTTPS Secure HyperText Transfer Protocol. A protocol for transferring encrypted hypertext data using Secure Socket Layer (SSL) technology.

JAD file Java Application Descriptor file. A file provided in a MIDlet suite that contains attributes used by application management software (AMS) to manage the MIDlet's life cycle, as well as other application-specific attributes used by the MIDlet suite itself.

JAR file Java Archive file. A platform-independent file format that aggregates many files into one. Multiple applications written in the Java programming language and their required components (`.class` files, images, sounds, and other resource files) can be bundled in a JAR file and provided as part of a MIDlet suite.

Java Community Process™ (JCP™) program Java Community Process program. An open organization of international developers and licensees who develop and revise Java platform specifications, reference implementations, and technology compatibility kits using a formal submission and approval process.

Java ME platform Java Platform, Micro Edition. A group of specifications and technologies that pertain to running the Java platform on small devices, such as cell phones, pagers, PDAs, and set-top boxes. More specifically, the Java ME platform consists of a configuration (such as CLDC or CDC) and a profile (such as MIDP or Personal Basis Profile) tailored to a specific class of device.

Java Specification Request (JSR) A proposal for developing new Java platform technology, which is reviewed, developed, and finalized into a formal specification by the JCP program.

Java Virtual Machine A software "execution engine" that safely and compatibly executes the byte codes in Java class files on a microprocessor.

KVM A Java virtual machine designed to run in small devices, such as cell phones and pagers. The CLDC configuration is designed to run in a KVM.

LCD Liquid Crystal Display. A common kind of screen display often used in small devices.

LCDUI Liquid Crystal Display User Interface. A user interface toolkit for interacting with LCD screens in small devices. More generally, a shorthand way of referring to the MIDP user interface APIs.

MIDlet An application written for MIDP.

MIDlet suite A way of packaging one or more midlets for easy distribution and use. Each MIDlet suite contains a Java application descriptor file (`.jad`), which lists the class names and files names for each MIDlet, and a Java Archive file (`.jar`), which contains the class files and resource files for each MIDlet.

MIDP Mobile Information Device Profile. A specification for a Java ME platform profile, running on top of a CLDC configuration, which provides APIs for application life cycle, user interface, networking, and persistent storage in small devices.

Obfuscation A technique used to complicate code by making it harder to understand when it is de-compiled. Obfuscation makes it harder to reverse-engineer applications and therefore, steal them.

Optional Package A set of Java ME platform APIs that provides additional functionality by extending the runtime capabilities of an existing configuration and profile.

PNG Portable Network Graphics. An image format commonly used with MIDP that can be compressed, transmitted, and stored without losing image quality.

Preemption Taking a resource, such as the foreground, from another application.

Preverification Due to limited memory and processing power on small devices, the process of verifying Java technology classes is split into two parts. The first part is preverification and done off-device using the preverify tool. The second part, which is verification, is done on the device at runtime.

Profile A set of APIs added to a configuration to support specific uses of a mobile device. Along with its underlying configuration, a profile defines a complete and self-contained application environment.

Provisioning A mechanism for providing services, data, or both to a mobile device over a network.

Push Registry The list of inbound connections, across which entities can push data, maintained by the Java Wireless Client software. Each item in the list contains the URL (protocol, host, and port) for the connection, the entity permitted to push data through the connection, and the application that receives the connection.

RMI Remote Method Invocation. A feature of Java SE technology that enables Java technology objects running in one virtual machine to seamlessly invoke objects running in another virtual machine.

RMS Record Management System. A simple record-oriented database that enables a MIDlet to persistently store information and retrieve it later. MIDlets can also use the RMS to share data.

SMS Short Message Service. A protocol allowing transmission of short text-based messages over a wireless network.

SOAP Simple Object Access Protocol. An XML-based protocol that allows objects of any type to communicate in a distributed environment, it is most commonly used to develop web services.

SSL Secure Sockets Layer. A protocol for transmitting data over the Internet using encryption and authentication, including the use of digital certificates and both public and private keys.

Sun Java Device Test Suite A set of Java programming language tests developed specifically for the wireless marketplace, providing targeted, standardized testing for CLDC and MIDP on small and handheld devices.

SVM Single Virtual Machine. A mode of the Java Wireless Client software, it can run only one MIDlet at a time.

task At the platform level, each separate application that runs within a single Java virtual machine is called a task. The API used to instantiate each task is a stripped-down version of the Isolate API defined in JSR 121. See the *CLDC HotSpot Implementation Architecture Guide* for more information.

TCP/IP Transmission Control Protocol/Internet Protocol. A fundamental Internet protocol that provides for reliable delivery of streams of data from one host to another.

WAE Wireless Application Environment. It provides an application framework for small devices, by leveraging other technologies such as WAP, WTP, and WSP.

WAP Wireless Application Protocol. A protocol for transmitting data between a server and a client (such as a cell phone) over a wireless network. WAP in the wireless world is analogous to HTTP in the World Wide Web.

WMA Wireless Messaging API. A set of classes for sending and receiving Short Message Service messages.

(x) button The button the user presses to end a task. On a real device this is the End key. On Windows it is the End key and sometimes the power key on the phone skin.

Index

A

all
  Java Wireless Client software build target, 40
  PCSL build target, 27, 29

C

clean
  Java Wireless Client software build target, 40
  PCSL build target, 27, 29
configuration options
  Java Wireless Client software
    SSL_DIR, 45
    SUBSYSTEM_LCDUI_MODULES, 43
    USE_CLDC_11, 42
    USE_MONET, 42
    USE_SSL, 45
  PCSL
    FILE_MODULE, 32
    MEMORY_MODULE, 32
    NETWORK_MODULE, 32
    PRINT_MODULE, 32

D

doc
  PCSL build target, 27, 29
docs_html Java Wireless Client software build target, 40
donuts
  PCSL build target, 29

F

FILE_MODULE
  PCSL configuration option, 32

J

Java Wireless Client software
  build targets, 40

M

MEMORY_MODULE PCSL configuration option, 32

N

NETWORK_MODULE PCSL configuration option, 32

P

PRINT_MODULE PCSL configuration option, 32

S

SSL_DIR Java Wireless Client software configuration option, 45
SUBSYSTEM_LCDUI_MODULES Java Wireless Client software configuration option, 43

T

targets
  Java Wireless Client software
    all, 40
    clean, 40
    docs_html, 40
  PCSL build targets
    all, 27, 29
    clean, 27, 29
    doc, 27, 29
    donuts, 29

U

`USE_CLDC_11` Java Wireless Client software configuration option, 42

`USE_MONET` Java Wireless Client software configuration option, 42

`USE_SSL` Java Wireless Client software configuration option, 45