



Build Guide

Sun Java™ Wireless Client Software 2.0 Java Platform, Micro Edition

Sun Microsystems, Inc.
www.sun.com

May 2007

Copyright © 2007 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, JavaCall, JDK, Javadoc, Java Developer Connection, Java Card, Java Community Process, JCP, HotSpot, Solaris, 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 Adobe 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 © 2007 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-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 plus des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etats - Unis et dans les 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 LAUTORISATION EXPRESSE, ECRITE ET PREALABLE DE SUN MICROSYSTEMS, INC.

Droits du gouvernement des États-Unis – logiciel commercial. Les droits de l'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 comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, JavaCall, JDK, Javadoc, Java Developer Connection, Java Card, Java Community Process, JCP, HotSpot, Solaris, 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 Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

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

OpenGL est une marque déposée de Silicon Graphics, Inc.

Le logo Adobe est 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 contrôle 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 manière 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 régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.



Adobe PostScript

Contents

| | |
|--|------------|
| Preface | vii |
| 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 the PATH | 6 |
| Verifying Windows Setup | 7 |
| Setting Up For the Build | 8 |
| Building the JavaCall API | 9 |
| Building PCSL | 10 |
| Building CLDC | 12 |
| Building Java Wireless Client Software | 13 |
| Running Java Wireless Client Software | 14 |
| 3. Quick Start: Building on Linux for Linux | 17 |
| Setting Up Your Environment | 17 |
| Building PCSL | 19 |

| | |
|--|-----------|
| Building CLDC | 20 |
| Building Java Wireless Client Software | 21 |
| Running Java Wireless Client Software | 22 |
| 4. Quick Start: Building on Linux for ARM | 25 |
| Setting Up Your Environment | 25 |
| Building PCSL | 27 |
| Building CLDC | 29 |
| Building Java Wireless Client Software | 30 |
| Running Java Wireless Client Software | 31 |
| 5. PCSL Build System | 33 |
| PCSL Build Overview | 33 |
| Output | 34 |
| Debugging Symbols | 34 |
| API Documentation | 34 |
| Selecting Modules | 35 |
| About Stubs | 36 |
| Building Individual Services | 37 |
| Network Service | 37 |
| Running Unit Tests | 38 |
| Extending the Build System | 38 |
| Creating a New Platform Makefile | 38 |
| Creating New OS and Compiler Makefiles | 39 |
| Creating a New Module | 40 |
| 6. CLDC Build System | 41 |
| CLDC Build Overview | 41 |
| Mapping Configuration Variables | 42 |

| | |
|--|-----------|
| 7. Java Wireless Client Software Build System | 43 |
| Overview | 43 |
| Output | 44 |
| Debugging Symbols | 45 |
| API Documentation | 45 |
| Build Options | 45 |
| CLDC Selection | 46 |
| Module Selection | 46 |
| Native AMS Image Resource Policy | 47 |
| Multitasking | 47 |
| Startup Performance | 47 |
| Resource Allocation Policy | 47 |
| Cryptography Selection | 48 |
| Server Socket Selection | 48 |
| Runtime Java Platform Properties Selection | 48 |
| Specifying a Target CPU and Device | 49 |
| Build Constraints | 49 |
| Including Optional APIs | 49 |
| Optional API Details | 50 |
| Building JSR 120 and JSR 205 | 51 |
| Building JSR 135 | 51 |
| Building JSR 177 | 51 |
| Building JSR 226 | 51 |
| Working With Stubs | 52 |
| Configuring the Build System for Stubs | 52 |
| Updating the Build System for Filled-in Stub Functions | 53 |
| Updating the Source Files and Build System after Porting | 54 |
| Glossary | 55 |

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 three chapters are designed to help you hit the ground running, so you can quickly navigate an entire build and run Java Wireless Client software.

[Chapter 2](#) illustrates how to build and run on Windows.

[Chapter 3](#) describes how to build and run on Linux.

[Chapter 4](#) shows how to build on Linux to run on an ARM device.

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

[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 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 | <i>directory></i> |

Typographic Conventions

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

Related Documentation

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

| Application | Title |
|--|--------------------------------|
| All | <i>Release Notes</i> |
| Porting | <i>Porting Guide</i> |
| Building | <i>Build Guide</i> |
| Using Tools | <i>Tools Guide</i> |
| Using Adaptive User Interface Technology | <i>Skin Author's Guide</i> |
| Using Multitasking Features | <i>Multitasking Guide</i> |
| Using MT-QTS | <i>MT-QTS Test Suite Guide</i> |
| Viewing reference documentation created by the Javadoc™ tool | <i>Java API Reference</i> |
| Viewing reference documentation created by the Doxygen tool | <i>Native API Reference</i> |

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>.

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 (Windows only).
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`, `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 use 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.

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 `docs/build-scripts/win32-x86`. 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 are incompatible in various ways and cannot be used without modifying the build system or source code.

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.

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\MICROS~2\VC\vcvarsall.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:\Applications\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 docs/build-scripts/win32-x86/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:\jwc\docs\build-scripts\win32-x86>setpath  
Setting environment for using Microsoft Visual C++ x86 tools.  
D:\jwc\docs\build-scripts\win32-x86>
```

Run the scripts in the rest of this chapter from the same 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.

In addition, the length of the paths you are using might also cause trouble in the build. Place the Java Wireless Client software source code in a location with a short path. The examples in this book are based on having the entire source code tree in `d:\jwc`.

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:\jwc\docs\build-scripts\win32-x86>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:\jwc\docs\build-scripts\win32-x86>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... ]
```

- **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:\jwc\docs\build-scripts\win32-x86>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)
D:\jwc\docs\build-scripts\win32-x86>javac
Usage: javac <options> <source files>
where possible options include:
...
D:\jwc\docs\build-scripts\win32-x86>which java
/cygdrive/d/j2sdk1.4.2_13/bin/java

D:\jwc\docs\build-scripts\win32-x86>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.

setup.sh looks like this:

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

This script is available as docs/build-scripts/win32-x86/setup.sh.

The Acme variable is the top level of the Java Wireless Client software. Adjust the value to point to the top level of your own installation. Don't run this script yet. It is 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 this 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 JAVACALL_PROJECT_DIR to the javacall-com directory.

The third variable, JAVACALL_OUTPUT_DIR, tells the build system where to put the output of the JavaCall API build.

You need to run the build from javacall-com/configuration/irbis/win32_x86.

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

```
. setup.sh

cd $Acme/javacall-com/configuration/irbis/win32_x86
make \
  JAVACALL_DIR=$Acme/javacall \
  JAVACALL_PROJECT_DIR=$Acme/javacall-com \
  JAVACALL_OUTPUT_DIR=$Output/javacall \
  $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\docs\build-scripts\win32-x86>sh build-javacall.sh
.
.
.
javautl_jad_parser.c
javautl_string.c
javautl_unicode.c
...Generating Library: d:/jwc/output/javacall/lib/javacall.lib
...compiling resources ...
d:\Applications\cyg4me\bin\make.exe:
javacall_status=OK

D:\jwc\docs\build-scripts\win32-x86>
```

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

```
D:\jwc\docs\build-scripts\win32-x86>ls d:\jwc\output\javacall
ext_lib  inc  lib  obj

D:\jwc\docs\build-scripts\win32-x86>
```

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++ or Visual Studio (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.

Tell PCSL where to put its output files with the PCSL_OUTPUT_DIR variable.

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

```
. setup.sh

cd $Acme/pcsl
make \
  JAVACALL_OUTPUT_DIR=$Output/javacall \
  PCSL_PLATFORM=javacall_i386_vc \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  $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\docs\build-scripts\win32-x86>sh build-pcsl.sh
.
.
.
building memory module...
d:\Applications\cyg4me\bin\make.exe[4]: Entering directory
`d:/jwc/pcsl/memory/heap'
d:\Applications\cyg4me\bin\make.exe[4]: Leaving directory
`d:/jwc/pcsl/memory/heap'
d:\Applications\cyg4me\bin\make.exe[3]: Leaving directory
`d:/jwc/pcsl/memory'
d:\Applications\cyg4me\bin\make.exe[2]: Leaving directory
`d:/jwc/pcsl/string/utf16'
d:\Applications\cyg4me\bin\make.exe[1]: Leaving directory
`d:/jwc/pcsl/string'
build-pcsl=OK

D:\jwc\docs\build-scripts\win32-x86>
```

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`.

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

- `JDK_DIR` is the location of the Java Development Kit software.
- `JVMWorkspace` is the `cldc` directory.
- The output of the CLDC build will be placed in `JVMBuildSpace`.

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

```
. setup.sh

cd $Acme/cldc/build/javacall_i386_vc
make \
    JDK_DIR=$JDK_DIR \
    ENABLE_PCSL=true \
    PCSL_OUTPUT_DIR=$Output/pcsl \
    JAVACALL_OUTPUT_DIR=$Output/javacall \
    JVMWorkspace=$Acme/cldc \
    JVMBuildSpace=$Output/cldc \
    $1

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

. $Scripts/teardown.sh
```

Run it with `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 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`.
- 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`.

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.

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

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

```
. setup.sh

cd $Acme/midp/build/javacall
make \
  JDK_DIR=$JDK_DIR \
  JAVACALL_PLATFORM=win32_i386_vc \
  JAVACALL_OUTPUT_DIR=$Output/javacall \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  CLDC_DIST_DIR=$Output/cldc/javacall_i386_vc/dist \
  TOOLS_DIR=$Acme/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 this script with `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.

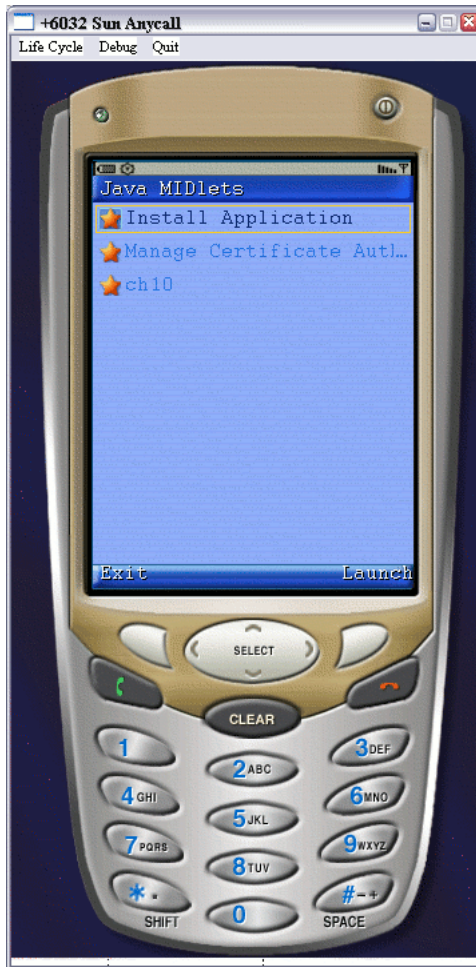
Running Java Wireless Client Software

Now what do you do with it?

To run the Java Wireless Client software stack in a simulated device, change directories to `output\midp\bin\i386`. Now run `usertest`.

A simulated device is displayed.

FIGURE 2-1 Simulated Device



You can install and test MIDlets on this simulated device. See the *Tools Guide* for complete details.

Quick Start: Building on Linux for Linux

This chapter describes a *self-hosted* Linux build, which means that you build and run the Java Wireless Client software on the same Linux machine. The goal of this chapter is to get you from zero to a functional MIDP software stack as fast as possible. Later chapters describe all the options for each part of the build.

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.
- Qt (qt) builds use the Qt toolkit directly. The high-level graphics system of Java Wireless Client software is implemented using Qt widgets. For this type of build, you need the Qt toolkit, available at <http://www.trolltech.com/products/qt>.

This chapter describes how to perform the `linux_fb_gcc` build.

This chapter is based on example scripts that help you run the different parts of the build. The scripts are available in `docs/build-scripts/linux-x86`. 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.

- **GNU Make**

```
$ 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.

■ **C/C++ compiler**

```
$ gcc --version
3.3.4
$
```

■ **JDK software**

```
$ java -version
java version "1.4.2_08"
Java(TM) 2 Runtime Environment, Standard Edition
(build 1.4.2_08-b03)
Java HotSpot(TM) Client VM (build 1.4.2_08-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 Acme=/home/jonathan/jwc
export JDK_DIR=/usr/java/j2sdk1.4.2_08
export Scripts=`pwd`
export Output=$Acme/output
export Log=$Acme/log.txt
rm -f $Log
```

This script is available as `docs/build-scripts/linux-x86/setup.sh`.

The `Acme` 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.

Don't run `setup.sh` yet. It is 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 this script.

Building PCSL

The first step is to build PCSL. Tell PCSL the target platform and where to put output files. Set `PCSL_PLATFORM` to `linux_i386_gcc`. Tell PCSL where to put its output files with the `PCSL_OUTPUT_DIR` variable.

In addition, tell PCSL what kind of networking implementation to use. Do this by setting `NETWORK_MODULE` to `bsd/generic`.

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

```
. setup.sh

cd $Acme/pcsl
make \
  PCSL_PLATFORM=linux_i386_gcc \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  NETWORK_MODULE=bsd/generic \
  $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:

```
build-scripts/linux-x86> sh build-pcsl.sh
.
.
.
building memory port module...
make[4]: Entering directory
`/home/jonathan/jwc/pcsl/memory/memory_port'
make[4]: Leaving directory
`/home/jonathan/jwc/pcsl/memory/memory_port'
building memory module...
make[4]: Entering directory
`/home/jonathan/jwc/pcsl/memory/malloc'
make[4]: Leaving directory
`/home/jonathan/jwc/pcsl/memory/malloc'
make[3]: Leaving directory `/home/jonathan/jwc/pcsl/memory'
make[2]: Leaving directory `/home/jonathan/jwc/pcsl/string/utf16'
make[1]: Leaving directory `/home/jonathan/jwc/pcsl/string'
build-pcsl=OK
build-scripts/linux-x86>
```

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

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

```
/home/jonathan> ls jwc/output/pcsl
linux_i386
/home/jonathan> ls jwc/output/pcsl/linux_i386
inc lib obj
/home/jonathan>
```

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` is the location of the output of the CLDC build.

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

```
. setup.sh

cd $Acme/cldc/build/linux_i386
make \
  JDK_DIR=$JDK_DIR \
  ENABLE_PCSL=true \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  JVMWorkSpace=$Acme/cldc \
  JVMBuildSpace=$Output/cldc \
  $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 `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_i386/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.

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

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

cd $Acme/midp/build/linux_fb_gcc
make \
    JDK_DIR=$JDK_DIR \
    PCSL_OUTPUT_DIR=$Output/pcsl \
    CLDC_DIST_DIR=$Output/cldc/linux_i386/dist \
    TOOLS_DIR=$Acme/tools
MIDP_OUTPUT_DIR=$Output/midp \
    $1

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

. $Scripts/teardown.sh
```

Run `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?

The `fb` build of the Java Wireless Client software uses a framebuffer for display and user input. Use the Qt tool `qvfb` to see the simulated screen of the device and provide keyboard input. `qvfb` supports up to ten independent displays, numbered from `:0` through `:9`.

Pick an available display and set `QWS_DISPLAY` to its number. Next, run `qvfb` with a bit depth of 16. Adjust the size of the framebuffer to match the expectations of the Java Wireless Client software.

```
export QTDIR=/opt/Qtopia
export QWS_DISPLAY=:4
$QTDIR/bin/qvfb -depth 16 -width 176 -height 210 &
```

Run `docs/build-scripts/linux-x86/run-qvfb.sh` if you wish. The `qvfb` window appears, but it is empty.

Now run the Java Wireless Client software. Set `QTDIR` and `QWS_DISPLAY` the same as before. Then run the `usertest` command.

```
export QTDIR=/opt/Qtopia
export QWS_DISPLAY=:4
output/midp/bin/i386/usertest
```

See `docs/build-scripts/linux-x86/run-usertest.sh` for an example.

The Java Wireless Client software runs in the `qvfb` window. Use your keyboard for input, if necessary. You might have to click in the `qvfb` window before using the keyboard. Use F1 and F2 for the soft buttons.

FIGURE 3-1 MIDlet Running in the `qvfb` Window.



You can install and test MIDlets on this simulated device. See the *Tools Guide* for complete details.

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.
- Qt (qt) builds use the Qt toolkit directly. The high-level graphics system of Java Wireless Client software is implemented using Qt widgets. For this type of build, you need the Qt toolkit, available at <http://www.trolltech.com/products/qt>.

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 `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.

- **GNU Make**
\$ `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`.

■ GCC cross compiler

```
$ gcc --version
3.3.1
$
```

■ JDK software

```
$ java -version
java version "1.4.2_08"
Java(TM) 2 Runtime Environment, Standard Edition
(build 1.4.2_08-b03)
Java HotSpot(TM) Client VM (build 1.4.2_08-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 Acme=/home/jk134021/jwc
export JDK_DIR=/usr/java/j2sdk1.4.2_08
export GNU_TOOLS_DIR=/opt/Embedix/tools/arm-linux
export Scripts=`pwd`
export Output=$Acme/output
export Log=$Acme/log.txt
rm -f $Log
```

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

The `Acme` 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 this script.

Building PCSL

The first step is to build PCSL. You have to do this twice to generate all the output that CLDC will need when you build it. First you need to build PCSL for `linux_i386_gcc`. Then you build PCSL for `linux_arm_gcc`.

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

cd $Acme/pcsl
make \
  PCSL_PLATFORM=linux_i386_gcc \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  NETWORK_MODULE=bsd/generic \
  $1

make \
  GNU_TOOLS_DIR=$GNU_TOOLS_DIR \
  PCSL_PLATFORM=linux_arm_gcc \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  NETWORK_MODULE=bsd/generic \
  $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:

```
build-scripts/linux-arm> sh build-pcsl.sh
.
.
.
building memory port module...
make[4]: Entering directory
`/home/jonathan/jwc/pcsl/memory/memory_port'
make[4]: Leaving directory
`/home/jonathan/jwc/pcsl/memory/memory_port'
building memory module...
make[4]: Entering directory
`/home/jonathan/jwc/pcsl/memory/malloc'
make[4]: Leaving directory
`/home/jonathan/jwc/pcsl/memory/malloc'
make[3]: Leaving directory `/home/jonathan/jwc/pcsl/memory'
make[2]: Leaving directory `/home/jonathan/jwc/pcsl/string/utf16'
make[1]: Leaving directory `/home/jonathan/jwc/pcsl/string'
build-pcsl=OK
build-scripts/linux-arm>
```

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

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

```
/home/jonathan> ls jwc/output/pcsl
linux_arm linux_i386
/home/jonathan> ls jwc/output/pcsl/linux_arm
inc lib obj
/home/jonathan>
```

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` just like for the PCSL build.

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

```
. setup.sh

cd $Acme/cldc/build/linux_arm
make \
  GNU_TOOLS_DIR=$GNU_TOOLS_DIR \
  JDK_DIR=$JDK_DIR \
  ENABLE_PCSL=true \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  JVMWorkSpace=$Acme/cldc \
  JVMBuildSpace=$Output/cldc \
  $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 `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` 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

cd $Acme/midp/build/linux_fb_gcc
make \
  GNU_TOOLS_DIR=$GNU_TOOLS_DIR \
  JDK_DIR=$JDK_DIR \
  PCSL_OUTPUT_DIR=$Output/pcsl \
  CLDC_DIST_DIR=$Output/cldc/linux_arm/dist \
  TOOLS_DIR=$Acme/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.

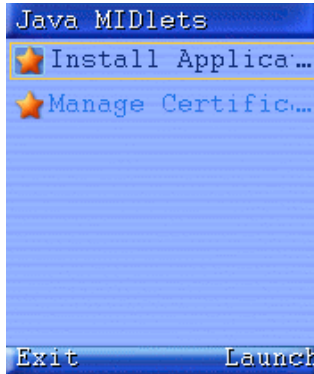
Running Java Wireless Client Software

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 4-1 Application Manager



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 |
|----------------------------|---|
| <code>all</code> (default) | Builds libraries and public header files in <code>\$PCSL_OUTPUT_DIR</code> . |
| <code>clean</code> | Removes all output for <code>\$PCSL_PLATFORM</code> from <code>\$PCSL_OUTPUT_DIR</code> . |
| <code>doc</code> | Builds API documentation in <code>\$PCSL_OUTPUT_DIR/doc</code> . |
| <code>donuts</code> | 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. bsd/qt (default) uses Qt/Embedded 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 as `-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.

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`.

Mapping Configuration Variables

Some CLDC HotSpot Implementation and Java Wireless Client software features much match.

Record the variables and values you use to build the CLDC HotSpot Implementation. You must use corresponding variables and values when you build the Java Wireless Client software.

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 6-1](#) lists build options in the two build systems.

TABLE 6-1 Configuration Options Mapping Between Build Systems

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

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: <code>\$ make parallel_all NUM_JOBS=4</code> |
| docs_html | Generates Porting API documentation. Define <code>DOXYGEN_CMD</code> 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 |
|--------------------------------|--|
| <code>USE_DEBUG</code> | 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. |
| <code>USE_I3_TEST</code> | Build an implementation with unit tests. Run all unit tests with <code>bin/i3test</code> command. |
| <code>USE_JAVA_DEBUGGER</code> | 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 <code>USE_DEBUG=true</code> . Example: <code>USE_JAVA_DEBUGGER=true</code> |
| <code>USE_JAVA_PROFILER</code> | 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. |

Module Selection

TABLE 7-4 shows the build options that control which LCDUI and AMS implementations are in the build.

TABLE 7-4 Module Selection Options

| Name | Value | Description |
|-------------------------|--|--|
| SUBSYSTEM_LCDUI_MODULES | chameleon (default) platform_widget | chameleon builds an implementation that uses adaptive user-interface technology. platform_widget uses Qt native widgets. |
| 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 7-5 shows the build options that control the cryptography configuration in the build.

TABLE 7-5 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_CRYPT0. |
| USE_RESTRICTED_CRYPT0 | true false (default) | Includes ciphers and the features that depend on ciphers, which are secure OTA, SecureConnection, HTTPS, and SATSA-CRYPTO (JSR 177). |
| RESTRICTED_CRYPT0_DIR | | Required when USE_RESTRICTED_CRYPT0=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_qt` and `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 `wince`, `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 `SUBSYSTEM_LCDUI_MODULES=platform_widget`, you must build using one of the `qt` configurations.
- 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_MULTIPLES_ISOLATES=true`, you must also set `SUBSYSTEM_LCDUI_MODULES=chameleon`.
- 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`.

Including Optional APIs

In general, optional APIs can be included by defining two variables. The first enables the API and has the form `USE_JSR_XXX`. The second specifies where to find the source code for the API and has the form `JSR_XXX_DIR`.

Most JSRs define a single API, but some contain more than one. JSR 75, for example, includes a PIM API and a FileConnection API. JSR 177 defines four distinct optional APIs. The build flags described in this section affect all the APIs defined in a JSR.

For example, if you want to include the JSR 75 PIM and FileConnection APIs, set `USE_JSR_75` to `true` and set `JSR_75_DIR` to `jwc/jsr75`, where `jwc` is the installation directory of the Java Wireless Client software.

Unless you specify otherwise, all the `USE_JSR_XXX` flags are `false`.

When you are building with the JavaCall API, you must provide identical sets of optional API flags to the JavaCall API build and the Java Wireless Client software build. For example, if you wish to build Java Wireless Client software with `USE_JSR_120=true`, you must first build the JavaCall API with `USE_JSR_120=true`.

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

- 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 238
- JSR 239

Some JSRs have more complicated builds. The next section describes how to build these optional APIs.

Optional API Details

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

Building JSR 120 and JSR 205

Unlike the JSRs, which can be built independently of each other, a dependency exists between the Wireless Messaging 1.0 optional API (JSR 120) and the Wireless Messaging 2.0 optional 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_JS_120` and `USE_JS_205` must both be set to `true`.

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

```
JSR_205_DIR=jwc/jsr205
```

Building JSR 135

Support for JPEG image format is a requirement of the MSA-248 Specification. Therefore, the JPEG decoder in your build is dependent upon the graphics platform used in your JSR 135 implementation.

For example, if you build a `linux_qte` configuration, the JSR 135 implementation can use the JPEG decoder from Qt. Therefore, no JPEG decoder is needed and the build option `USE_JPEG=false` is set.

On the other hand, if you build a `linux_fb` configuration, the JSR 135 implementation does not contain its own JPEG decoder and must be picked up from somewhere else. Therefore, set `USE_JPEG` to `true` and `JPEG_DIR` to the JPEG codec source code directory, most likely `jwc/jpeg`.

Building JSR 177

To properly build the Security and Trust Services APIs (JSR 177) optional package, you must set the `USE_JS_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
```

Building JSR 226

The JSR 226 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, usually `jwc/pisces`.

JSR 226 uses the XML parser that is part of JSR 172 (Web Services). If you plan to include JSR 226, you must also include JSR 172.

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.

For more information, see the *Porting Guide*.

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. Use `build/linux_qte_gcc/qte.gmk` as a template for creating `stubs.gmk`.

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 copying, 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 copying, 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, 44
- PCSL build target, 33

C

clean

- Java Wireless Client software build target, 44
- PCSL build target, 33

configuration options

- Java Wireless Client software
 - SSL_DIR, 48
 - SUBSYSTEM_LCDUI_MODULES, 46
 - USE_CLDC_11, 46
 - USE_MONET, 46
 - USE_SSL, 48

PCSL

- FILE_MODULE, 36
- MEMORY_MODULE, 36
- NETWORK_MODULE, 36
- PRINT_MODULE, 36

D

doc

- PCSL build target, 33

docs_html Java Wireless Client software build target, 44

donuts

- PCSL build target, 33

F

FILE_MODULE

PCSL configuration option, 36

J

Java Wireless Client software
build targets, 44

M

MEMORY_MODULE PCSL configuration option, 36

N

NETWORK_MODULE PCSL configuration option, 36

P

PRINT_MODULE PCSL configuration option, 36

S

SSL_DIR Java Wireless Client software
configuration option, 48

SUBSYSTEM_LCDUI_MODULES Java Wireless Client
software configuration option, 46

T

targets

Java Wireless Client software

- all, 44
- clean, 44
- docs_html, 44

PCSL build targets

- all, 33
- clean, 33
- doc, 33
- donuts, 33

U

USE_CLDC_11 Java Wireless Client software
configuration option, 46

USE_MONET Java Wireless Client software
configuration option, 46

USE_SSL Java Wireless Client software
configuration option, 48