Revolution SDK

Build System

Version 1.04

2009/11/25

The contents in this document are highly confidential and should be handled accordingly.

RVL-06-0044-001-E
Released: February 3, 2010

Contents

Code Examples

Figures

Revision History

Version	Revision Date	Item	Description
1.04	2009/11/25	Entire document	Standardized captions throughout document (Japanese version only) Changed descriptions related to CodeWarrior to be in-line with CodeWarrior for Wii Changed Metrowerks to Freescale.
		1	Changed description related to Nintendo GameCube.
		Figure 1	Changed overview content.
1.03	2007/07/20		Merged with the Build System document from Nintendo GameCube.
1.02	2006/08/15	3	Revised the methods for running demos and debugging.
1.01	2006/04/07	3.2	Added a note specific to debugging the samples.
1.00	2006/03/22	-	First release by Nintendo of America, Inc.

1 Overview

This document provides an overview of the build system for the Revolution SDK.

2 Assumptions

This document assumes that you have successfully configured the hardware and installed the components as described in the Quickstart Guide for the Revolution SDK Development Environment (`RVL-QuickStartGuide.us.pdf`). This includes the following.

- Cygwin Tools (including GNU make 3.80)
- CodeWarrior for Wii
- Revolution SDK
- NDEV Disc Emulation System

For brevity's sake, we assume that you have installed Revolution SDK under the `/RVL_SDK` directory in the examples throughout this document.

This document also assumes a basic working knowledge of the GNU make tool.

3 Running Demos

3.1 Initializing the Revolution Shell

The batch file, `RVL_NDEV.bat`, initializes the environment variables and calls a Cygwin bash shell. To build or debug, use `RVL_NDEV.bat` to open a shell.

Take note that the environment variables are valid as local settings only for the duration of the shell session (and any child sessions). These settings are neither permanent nor will they effect other areas of the system. As a result, multiple build environments, such as those for the Nintendo DS™ and the Nintendo GameCube, can exist on the same PC.

Be aware of the following.

- You should only call the Revolution SDK build system from this shell. Build systems for other platforms are not guaranteed to work correctly.
- The shell only supports the Bash protocol.

3.2 Running a Demo from the Command Line

Open a Revolution shell. From the command line, type:

```
% cd /RVL_SDK/RVL/bin/demos/gxdemo
% ls
```

The `*.elf` files are executables that run on NDEV. Debug versions have the suffix `*D.elf`.

To run one of these executables (for example, smp-onetriD.elf), simply type:

```
% ndrun smp-onetriD.elf
```

3.3 Running a Demo from the Debugger

To run the executable from the CodeWarrior IDE/debugger, it must be called within the Revolution shell.

The CodeWarrior IDE/debugger is unique to Wii. It must be called from this shell session. To launch the IDE/debugger, enter the following:

```
%rvl_ide.sh
```

Once it launches, Revolution project files and ELF files can be dragged and dropped into the debugger.

If this is the first time you are debugging this particular application, the IDE will generate a project for it. This will take a moment as the debugger locates all of the sources referenced in the executable.

The project file is stored as an `.mcp` file in the same directory as the application. If you delete this .mcp file, all of your project settings will be lost.

The pre-built demos included in Revolution SDK cannot be debugged as is. To debug, you must rebuild each in your own environment.

4 Building a Sample Application

The Revolution SDK includes the *samplebuild* module as an example of a simple application and library. From the command line, enter:

```
% cd /cygdrive/c/RVL_SDK/build/samplebuild
```

This is a sample application which includes the library required by the application. Developers can use this a framework to start a new project.

To build from the command line, enter:

```
% make
```

4.1 Platforms

Note that the default hardware platform is RVL. Other platforms are for internal use only and not supported. However, future revisions of the NDEV system may require redefinition of the PLATFORM flag to build hardware-appropriate code.

To explicitly define the platform, type:

```
% make PLATFORM=RVL
```

4.2 DEBUG and NonDebug Builds

The default target is DEBUG, in which the build is compiled without optimizations. Furthermore, all of the system libraries perform additional sanity checking. Note that debug symbolics are NOT included in the system libraries.

To explicitly build a DEBUG target, type:

```
% make DEBUG=TRUE
```

To build a release (optimized) version, type:

```
% make NDEBUG=TRUE
```

For NDEBUG builds, the compiler performs optimizations but only file-level interprocedural analysis, biasing for speed rather than code size. Note that debug symbolics are not included.

4.3 Cleaning

To remove all objects, dependency files, and disassembly files, type:

```
% make clean
```

This will clean the samplebuild project for both DEBUG and NDEBUG targets.

To remove all executables and libraries, type:

```
% make clobber
```

To remove all executables and libraries by folder, type:

```
% make superclobber
```

4.4 Building Projects

The library associated with the samplebuild module will be located here.

`/RVL_SDK/build/samplebuild/sample/lib/RVL/samplelibD.a`

The executable will be located here.

`/RVL_SDK/build/samplebuild/sample/bin/RVL/samplebinD.elf`

Note that both the library and executable have `*D.a` and `*D.elf` suffixes, indicating that they are *debug* versions.

Non-debug versions will appear in the same directories, and will not have the “D” suffix.

Also note that the `bin/` and `lib/` directories are further organized by hardware platform. In this example, all build products are placed under the `RVL/` directory.

4.5 Building Individual Executables

To explicitly build a given executable, you can type:

```
% make bin/RVL/samplebinD.elf
```

This will build the debug version of the samplebin executable.

To build the non-debug version, type:

```
% make bin/RVL/samplebin.elf
```

Note that the appropriate (debug or non-debug) libraries must have already been built for this to work.

4.6 Adding Files

In this example, we'll add source files to the src/ directory of the samplebuild module. One file will be added to the executable; the other to the library.

Ensure that you are in:

/RVL_SDK/build/samplebuild/sample/src/

Create the files:

```
% touch newlibfile.c
% touch newbinfile.c
```

You can leave the files empty.

Open the makefile in a text editor:

/RVL_SDK/build/samplebuild/sample/makefile

Add newlibfile.c to the CLIBSRCS variable:

```
CLIBSRCS = samplelib.c newlibfile.c
```

This informs the build system that these two files should be built and linked into the library specified by LIBNAME (samplelib in this case). Note that if LIBNAME is undefined, no library is created.

Add newbinfile.c to the CSRCS variable:

```
CSRCS = samplebin.c newbinfile.c
```

This informs the build system that these two files should be built. It does not indicate which executables they should be linked into. That's done with the dependency line at the bottom of the makefile. Modify the dependency line to add the new object file:

```
$(FULLBIN_ROOT)/samplebin$(BINSUFFIX): samplebin.o/
newbinfile.o/
lib/$(ARCH_TARGET)/samplelib$(LIBSUFFIX)/
$(REVOLUTION_LIBS)
```

This dependency tells the build system when to re-link samplebin, and what should be linked. Note that the build system will only look at this dependency if the binary name samplebin has been added to the BINNAMES variable.

Now, invoke a build:

```
$ cd /RVL_SDK/build/samplebuild/sample
$ make
```

You will see the build system compile and link the newly added files.

To see a simple example of how multiple executables can be created from one makefile, examine the demo:

/RVL_SDK/build/demos/wpaddemo

4.7 Creating Your Own Modules

The simplest way to create a new module is to copy an existing samplebuild module.

From the bash shell:

```
cp -r /RVL_SDK/build/samplebuild/sample /RVL_SDK/build/samplebuild/mymodule
```

This creates a new module mymodule at the same directory level as sample.

Open the makefile in mymodule/ with a text editor, and change the MODULENAME variable:

```
MODULENAME = mymodule
```

This variable simply indicates where in the samplebuild tree this module exists. The build system knows this module is in the samplebuild tree because the variable SAMPLE is set to TRUE.

If we are creating a new library that will be linked in by other modules, the library must be installed into a public location. This is achieved by adding the install target to the all target:

```
all:setup build install
```

Each of these three targets is defined by the build system.

The install target will make the build system copy the library into /RVL_SDK/RVL/lib. Other executables that wish to link in the library will simply include it in their dependency line, like so:

```
/RVL_SDK/$(ARCH_TARGET)/lib/samplelib$(LIBSUFFIX)
```

Note that the use of $(ARCH_TARGET) and $(LIBSUFFIX) allow the build system to use the same dependencies for different kinds of builds.

The library name can be changed by modifying the LIBNAME variable. You should not add any suffix, as the build system does that automatically to reuse the same root names across different builds. The real name of the library for a particular build is always:

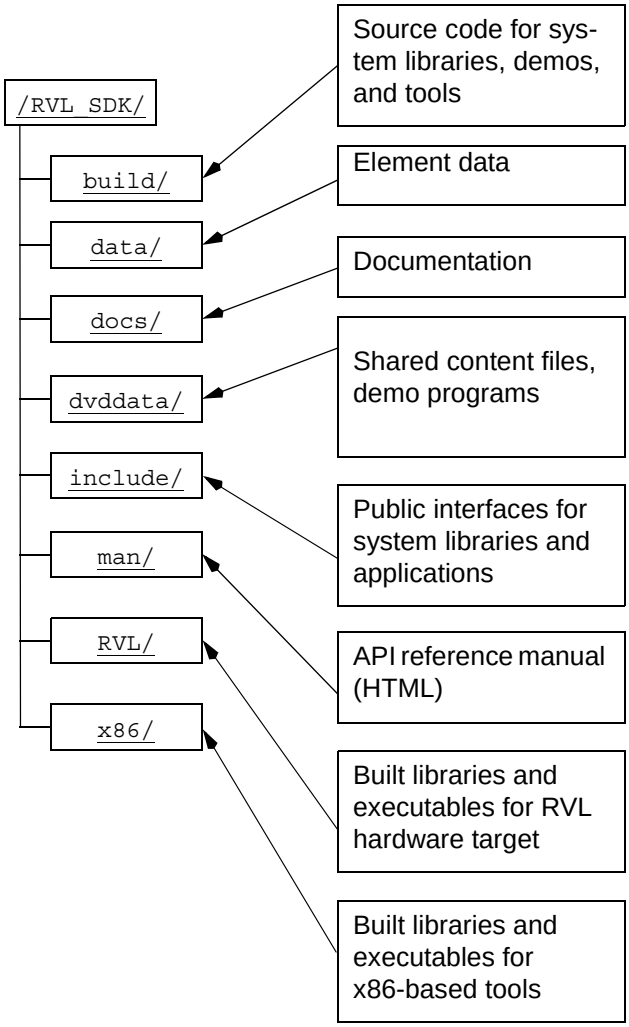
$(LIBNAME)$(LIBSUFFIX)

The executable name can be changed by modifying BINNAMES and the corresponding dependency line at the bottom of the file.

5 Build Tree Overview

The figure below illustrates the high-level structure of the build tree.

Figure 5–1 Top-level SDK Overview



5.1 Interfaces

To access all system libraries, include the following header file:

`/RVL_SDK/include/revolution.h`

This header file in turn includes module-specific interfaces. If you need to access specific modules, the associated header files are located under:

`/RVL_SDK/include/revolution`

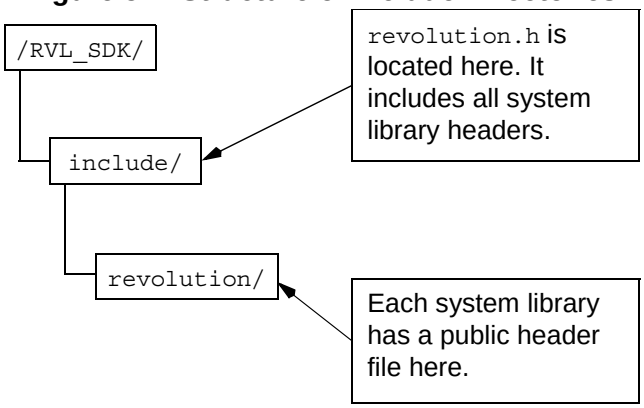
For example:

```
#include <revolution.h> // include all library headers
```

or

```
// include just OS and Graphics interfaces
#include <revolution/os.h>
#include <revolution/gx.h>
```

Figure 5–2 Structure of Include Directories



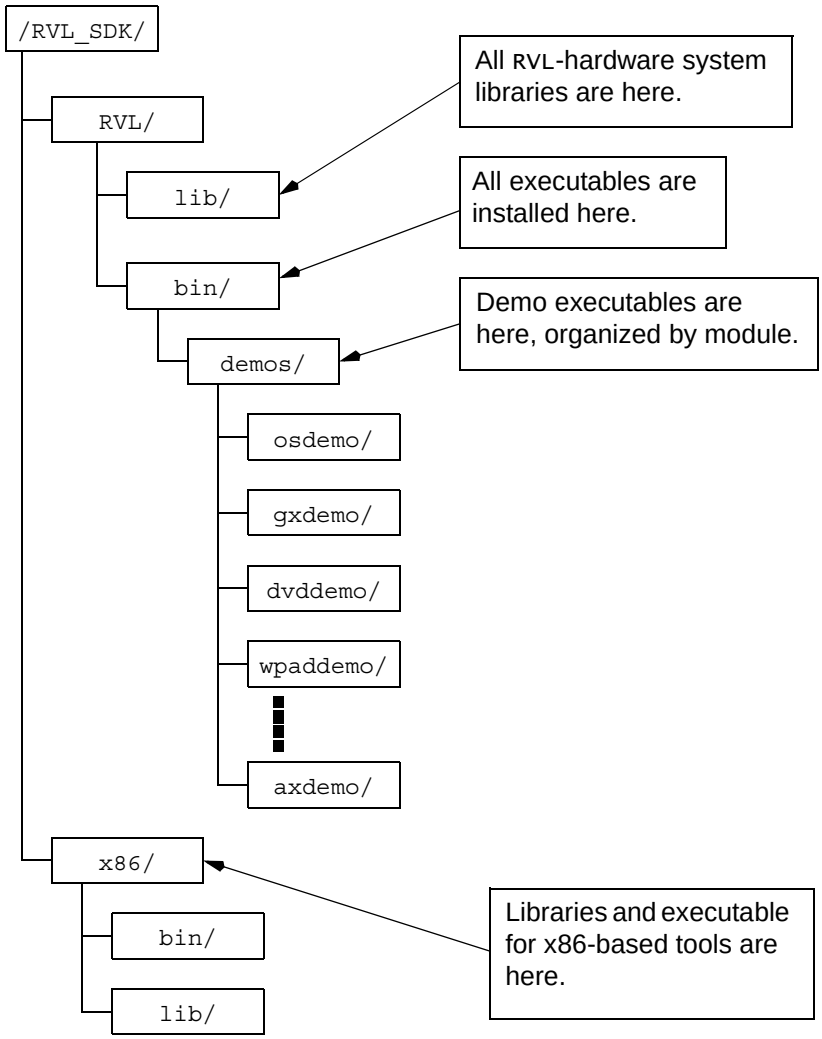
Note that the `/RVL_SDK/include//` directory is not shown. This directory contains module header files that simply point to the corresponding file under `revolution/`.

These header files are meant to facilitate migration of Nintendo GameCube applications to Wii.

5.2 Building Projects

Libraries and executables are organized by hardware platform, relative to the top-level directory of the SDK.

Figure 5–3 Libraries and Executables



5.2.1 Naming Rules for Binary Files

System library names are expressed as `{modulename}{D if debug}.a`. For example, `os.a` is a non-debug, optimized version of the OS library, and `osD.a` is the debug version.

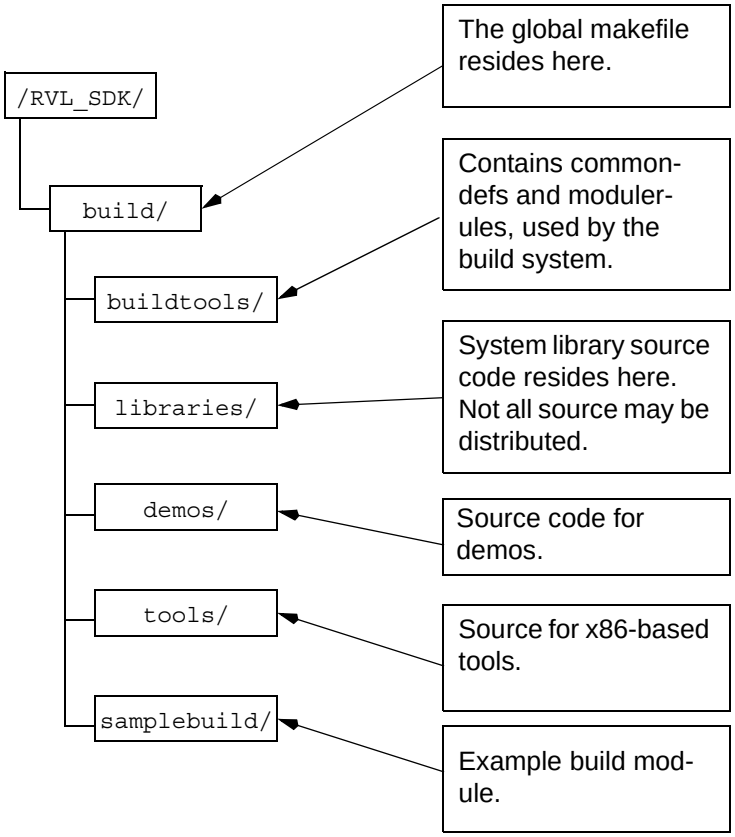
The suffix `.elf` is appended to names of executable programs like demo applications. `D` will also be added in the case of a debug version. For example, the optimized version of the `onetri` demo in GX is `onetri.elf`, and the debug version is `onetriD.elf`.

Debug versions of executable programs only use the debug versions of libraries and object files, and optimized versions of executable programs only use the optimized versions of libraries and object files.

5.3 Source Code

Source code for all libraries, demos, and tools are located under the `/RVL_SDK/build/` directory. Note that not all system libraries are accompanied by source code.

Figure 5–4 Source Tree

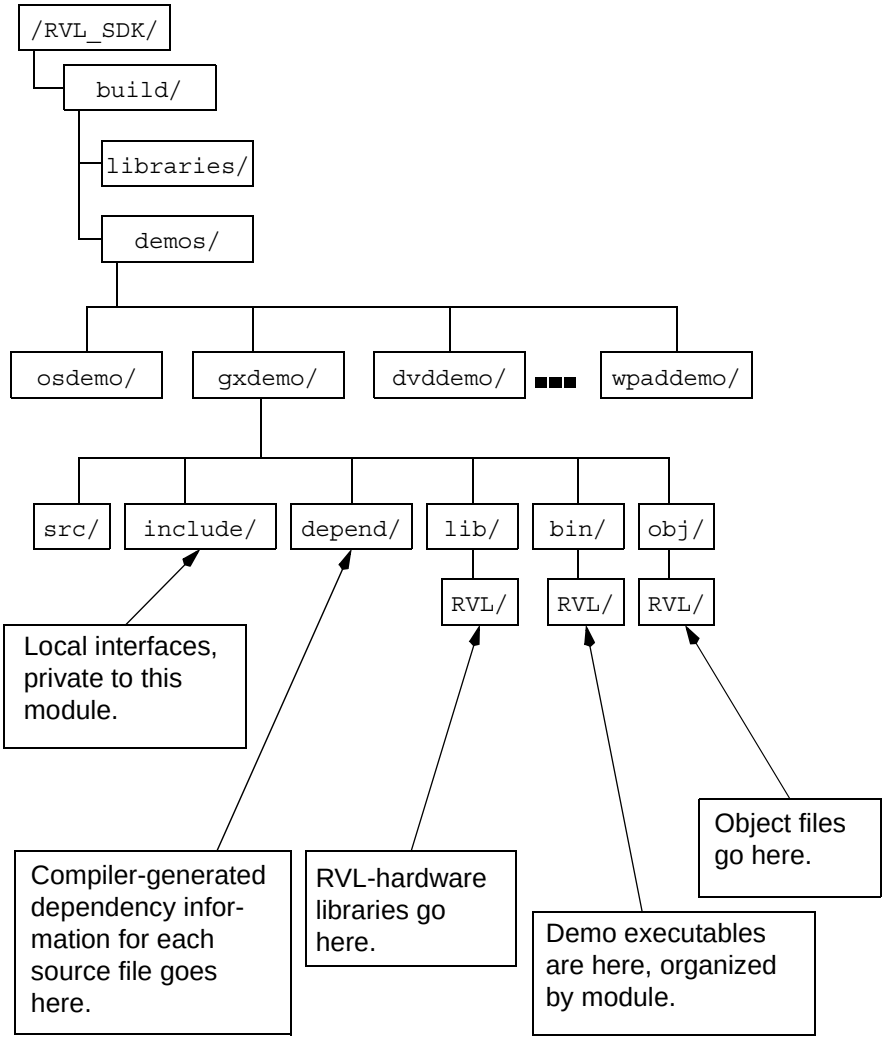


The most important files within these directories are `buildtools/modulerules` and `buildtools/commondefs`. These are critical, so we have a separate chapter for them in this document. Refer to “Operation of commondefs and modulerules” on page 27, for details.

5.4 Demo Modules

Demo modules are organized by programming guide (see the `/RVL_SDK/docs`) and may include one or more applications. Local build products are stored in `lib/`, `bin/`, and `obj/` directories, organized by hardware platform.

Figure 5–5 Demo Module Structure



Each module has its own `makefile` at the root of its subtree (For example: `/RVL_SDK/build/demos/gxdemo/makefile`). These `makefiles` are extremely simple, and rely on the `commondefs` and `modulerules` files in `/RVL_SDK/build/buildtools` to generate dependencies and to run the compiler and linker.

The local `include` directory under each module is meant for special header files that cannot be shared between modules. The demos do not use these kinds of header files, but the system library makes frequent use of them.

The same trees are used to build code with multiple targets, so an "`RVL`" directory must be present under the `lib`, `bin`, and `obj` directories.

6 Compiler

This chapter describes the CodeWarrior compiler that is provided along with the SDK and used by the build system. It also describes in detail cautions about using the compiler in this environment.

6.1 Overview

This distribution uses an embedded PowerPC cross-compiler that runs on x86 systems.

PowerPC code generation and optimization are complete and stable, but the operations and arguments for the command-line interface are different from those of most widespread UNIX compilers.

If you wish to create your own build system, refer to this document and `/RVL_SDK /build/build-tools/commondefs`, where the appropriate compiler arguments have been documented.

6.2 CodeWarrior Setup

The following components are installed on the PC (by default, all are installed in the `C:\Program Files\Freescale\CW for Wii v1.0` directory.

- CodeWarrior embedded PowerPC cross-compiler
- Support Libraries (in `Freescale\CW for Wii v1.0\PowerPC_EABI_Support\Runtime`)
- Standard C libraries (in `Freescale\CW for Wii v1.0\PowerPC_EABI_Support\MSL`)
- CodeWarrior IDE
- Documentation

Note: EABI is an abbreviation for “Embedded Application Binary Interface.”

CodeWarrior PowerPC Cross-Compiler – Comprised of the PowerPC cross-compiler, assembler, and linker. These are all located in `CW_for_Wii v1.0\PowerPC_EABI_Tools\Command Line Tools` in the CW for Wii v1.0 directory. With regard to how to use the command-line tools, this document basically only covers the procedure for using the compiler and a reference for command-line options. Refer to the following three locations as references for the command-line options.

- The appendices of this document
- `CW for Wii v1.0\PowerPC_EABI_Tools\Command_Line_Tools\*.txt`
- The `-help` switch at the command line

Support Libraries: Several Freescale support libraries are required for the application to run properly. These are included transparently by means of the `$(EPPCMWLIBS)` variable (set in the `commondefs` file). There is no need to add any separate libraries.

CodeWarrior IDE: The IDE is mostly used for debugging, and controls the operation of NDEV. The IDE can also be used for source editing and version control.

Documentation: The documentation for each of the various CodeWarrior components can be found in `CW for Wii v1.0\PowerPC_EABI_Support\Documentation` and `CW for Wii v1.0\CW Release Notes`. The “IDE Guide” and “C Compilers Reference” are particularly important. The former deals with how to use the debugger and editor, and the latter deals with compiler-specific C language.

7 Building the Libraries and Demos

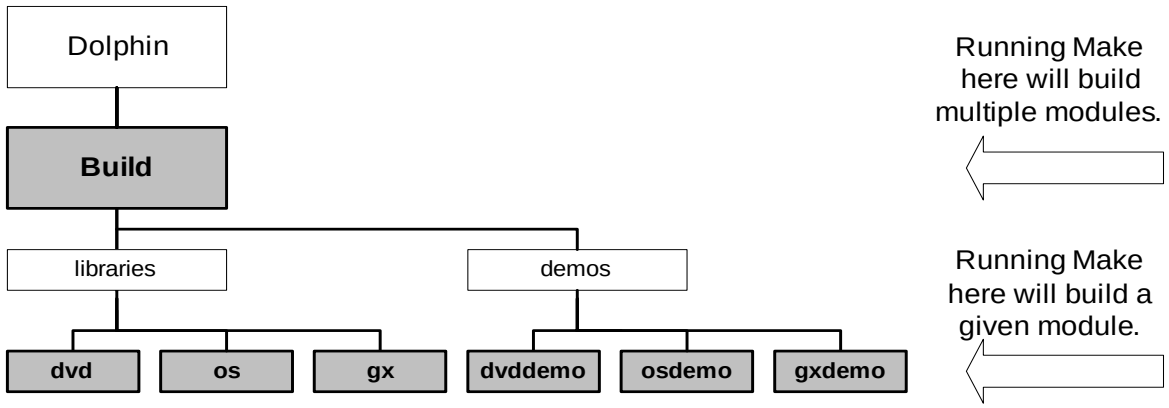
This chapter explains in detail how to build the libraries and demo programs with this build system. The sections below explain in detail how the build system actually operates.

7.1 Build Environment

Refer to the *Quickstart Guide* (`RVL-QuickStartGuide.en.pdf`).

7.2 Running make

Figure 7–1 Places Where `make` Can be Run Safely



This document assumes that you will be using the Cygnus bash shell to perform builds. Although it is *possible* to run Make from the MS-DOS shell (by typing "`make --unix`"), there is probably no reason to go out of your way to use the MS-DOS shell.

The two main locations where `make` is run are the following:

- **`/build`**: This will use the global makefile used for building the entire system (the libraries and demos). Running Make here will build all libraries and demos. The Make command will skip modules without source.
- **`/build/{libraries|demos}/{modulename}`**: This will use the makefiles for each local module in order to build a given module.

Notification will be made when a makefile is moved from one module to another, when compilation of the source files begins as well as when object files and libraries have been linked with the executable programs. When the build of a module has completed, the makefile will copy the libraries and binary files that were built from the local `lib` and `bin` directories to the `lib` and `bin` directories in `/RVL`.

7.3 Targets for DEBUG and Non-Debug (Optimized) Versions

Only DEBUG targets are built by default. In other words, compiler optimization will not be performed. All system libraries run an additional validity check (by means of the `ASSERT`/`ASSERTMSG` macros and external validity-checking code).

In order to perform a completely-optimized build, type "`make NDEBUG=TRUE`". Optimized targets will enable all compiler optimizations. This optimizes speed, not code size, so the line numbers in the debugger will no longer be correct. Symbol information will be generated, but several functions will turn into inline functions, and the configuration may be changed a lot, so debugging will be extremely difficult.

7.4 Building a Given Binary File

If you know the name of the binary file that you want to build, you can build from the makefile level of that module. For example, in order to build `onetriD.elf`, type "`make onetriD.elf`" in `/build/demos/gxdemo`.

Note that the name of the binary file reflects whether or not to perform a DEBUG build. For example, in order to build the optimized version of `onetri.elf` (note that there is no "`D`" appended), you must type "`make NDEBUG=TRUE onetri.elf`". Running "`make onetri.elf`" will not work because "`onetri.elf`" will not be built during a DEBUG build.

7.5 Deleting Binaries

In order to delete all objects related to a target, type "`make clean`". This will delete all `.o` files, dependency files, and disassembly files. This applies both to `DEBUG` and `NDEBUG` targets.

In order to delete all executable programs and libraries, type "`make clobber`".

7.6 Disassembly

Disassembly is useful for examining code generated by the compiler without running the debugger. It is also useful for finding the memory addresses of symbols and variables.

All the CodeWarrior command-line tools in the execution path are useful for disassembling files. Just add the "`-dis`" option and run the linker, then pipe the output to a file as shown below.

```
mwldeppc -dis -proc gekko onetriD.elf > onetriD.disassemble
```

Gekko-specific op-codes may be output during disassembly, so "`-proc gekko`" is required.

Note: Although the code name for the CPU installed in the Wii is *Broadway*, specify "`gekko`" as an compiler argument.

8 Running Applications

The following are the three main ways to run an application on NDEV.

1. Run it directly from the command line. This is explained in Chapter 3.
2. Run it by starting the debugger. This is explained in Chapter 3.
3. Run it within the edit-debug cycle.

8.1 Workflow of the Edit-Debug Cycle

It is assumed that the debugger has already been installed, and the Project window of the application to debug is opened.

1. Debug the application.
2. Find a bug.
3. Stop the current debug session. If a debug session is in progress (in other words, if CodeWarrior is connected to NDEV), the executable program will be locked, and the build will fail during linking. There is no need to exit the CodeWarrior IDE.
4. Edit the source.
5. Rebuild the application.
6. Go back to step 1. You won't have to double-click on the `.elf` file again, nor will you have to restart CodeWarrior or recreate the project file.

For details on how to use the debugger, especially with regard to OS contexts, refer to "How to Debug" in this guide.

9 Makefile Operation

We haven't covered how to use makefiles up until this point. This chapter explains module-level makefiles in detail. This information makes it possible for you to create your own modules and libraries within the infrastructure that has been provided. Appendix A explains in detail how the build system actually works in order to assist you in creating your own build system.

9.1 Overview

The following three components are required to build a module.

- **Makefiles for local modules**: The local makefiles for each module define what type of binary file to build and define each source file.
- **`commondefs`**: This file defines all paths and compiler flags used during the build. It also controls several libraries that are linked to the executable program.
- **`modulerules`**: This file contains all of the general rules for deciding what kind of build will be done. It also generates dependencies, sets up directories, and deletes binary files.

Each of these files are well-documented and can be understood with just a minimal knowledge about GNU Make. This section starts off with modules' makefiles, then goes on to explain in detail the most important parts of each of the three components. It explains how they work together and gives some examples about compiler flags and other issues that should be considered.

9.1.1 Paths

When building a makefile in Windows, how to use the path is important. External environment variables (for example, the Windows environment) will likely use backslashes as delimiters for directories. However, the backslash has a different meaning in UNIX shells and GNU Make. The normal way of handling spaces within path names in Windows (that is, double quotes) is also undesirable for GNU Make. Follow the guidelines shown below:

- Replace all backslashes (\) with forward slashes (/).
- Add a backslash before all spaces within the path (for example, `Program\ Files`).
- Do not use quotes anywhere. Doing so will probably cause problems with the GNU Make tool, the shell, and the compiler.
- Select the type of output path for the compiler (for example, when using the "`-o`" option). Use a relative path for the output path (in other words, do not begin it with a slash). This is not a problem for the linker.
- As is the case for library paths and include paths, both absolute paths and relative paths can be used for objects to link and source code to compile.

9.2 Makefiles for Modules

This section uses `/build/samplebuild/sample/makefile` in order to explain, in order, the basics for specifying libraries and executable programs with this build system.

With this build system, a maximum of one library and an arbitrary number of executable files can be built for a local module. A makefile starts off with a header like the one in "Code 9–1 Makefile Header" on page 23.

Code 9–1 Makefile Header

```
all:    setup build

# commondefs must be included near the top so that all common variables
# will be defined before their use.
include $(REVOLUTION_SDK_ROOT)/build/buildtools/commondefs

# MODULENAME should be set to the name of this subdirectory
MODULENAME     = sample

# This indicates to commondefs/modulerules as to where this directory is
# The selectable paths are restricted to limit the number of subtrees
# that are added to the  tree.
SAMPLE         = TRUE
:
include $(REVOLUTION_SDK_ROOT)/build/buildtools/modulerules
:
```

all: All modules must have "setup" and "build" targets. The "setup" target merely verifies that the directory structure has been prepared, and the "build" target actually builds the source and performs linking. The system libraries and demos also contain an "install" target, which copies the libraries that were built into the binary tree for the given architecture target (/RVL in this case).

commondefs: A commondefs file must be included at the beginning to configure general variables for the overall makefile.

MODULENAME: The current module name shows the subdirectory names. The build system is able to create the correct path for this module using this and the variables shown below.

SAMPLE = TRUE: Indicates that the current tree is in "/build/samplebuild". This method was chosen instead of directly specifying the tree with "THISTREE = samplebuild" so as not to extend unnecessary tree structures. The following options are also available:

- LIB: System libraries
- DEMO: Demos
- TEST: System test code
- CHAR_PIPELINE: Character pipeline library code

modulerules: The modulerules file contains the actual building and linking rules.

9.2.1 Specifying Libraries to Build

Specify which libraries to build using the following three variables.

Code 9–2 Library Build Variables

```
LIBNAME = samplelib

CLIBSRCS= samplelib.c
ASMLIBSRCS  =
```

LIBNAME: The name of the library. Do not include a suffix in this name, as the library's suffix will depend on whether it is a debug build (refer to "5.2.1 Naming Rules for Binary Files" on page 15, for details about the naming rules). The final name of the library will be $(LIBNAME)$(LIBSUFFIX), using the LIBSUFFIX that is configured in commondefs.

CLIBSRCS: A list of all the C source files to build and link with the given library.

ASMLIBSRCS: A list of all the assembly source files to build and link with the given library. Generally, these will not be pure assembly files, but rather inline assembly within C code (using `asm` keywords).

9.2.2 Specifying Executable Programs to Build

Specify how to build executable programs with a set of similar variables, as shown below. The dependencies for each library also must be specified in the makefile.

Code 9–3 Build Variables for Executable Programs

```
BINNAMES        = samplebin

CSRCS           = samplebin.c
ASMSRCS         =
:
:
$(FULLBIN_ROOT)/samplebin$(BINSUFFIX): samplebin.o \
                        lib/$(ARCH_TARGET)/samplelib$(LIBSUFFIX) \
                        $(LIBS)
```

BINNAMES: A list of the names of the executable programs to link. As long as there are rules in place for the various dependencies that specify which object files to link, an arbitrary number of executable programs can be listed here.

CSRCS: A list of the C source files to build. Generally, this is a list of all the C files to link with the executable program.

ASMSRCS: A list of the assembly source files to build. Generally, these will not be pure assembly files, but rather inline assembly within C code (using `asm` keywords).

9.2.2.1 Executable Program Dependencies

Dependency information for executable programs is listed at the end of the file. The standard rule format for `make` is as follows:

Code 9–4 Make Rule Format used to Specify Objects to Link with Executable Programs

```
$(FULLBIN_ROOT)/<executable name>$(BINSUFFIX): <objects to be linked>
```

$(FULLBIN_ROOT) specifies the directory in which to create the binaries (in this case, `/build/samplebuild/sample/bin/RVL`). This must be a full path, because GNU Make matches the target names exactly.

$(BINSUFFIX) is "`D.elf`" for debug builds or "`.elf`" for optimized builds.

$(LIBS) includes all system libraries required by the application.

Local Library: The sample makefile also links `samplelib` in the local library to build. This library is accessed using `lib/$(ARCH_TARGET)/samplelib$(LIBSUFFIX)` in the location to build. The library and executable program are always referenced as `<libname>$(LIBSUFFIX)` or `<binname>$(LIBSUFFIX)`, respectively. The same rule applies both to debug versions and optimized versions.

Compiler-Exclusive Libraries: The build system automatically links all libraries such as the standard C runtime library with the application. The linker will delete all dead code, so you needn't worry about the size increasing due to unnecessary code.

10 Adding Your Own Modules

Once you have understood how the makefiles work, you will probably want to create your own module trees. The guidelines for several necessary steps are described below.

- Create the build tree (Example: `/build/MyApplication`)
- Copy an existing module tree, such as `/build/samplebuild/sample`, and create a subtree for the module (Example: `/build/MyApplication/MyFirstModule`)
- Add the build tree to `modulerules` as shown below.

Code 10–1 Adding a Build Tree to modulerules

```
ifeq ($(MYAPPLICATION), TRUE)
PROJ_ROOT      = /build/MyApplication
endif
```

- Edit the `makefile` of the local module as follows (overwrite "`MODULENAME=SAMPLE`" and "`SAMPLE=TRUE`")

Code 10–2 Editing the makefile of the Local Module

```
MODULENAME=MyFirstModule
MYAPPLICATION=TRUE
```

- Add the source
- Correctly change the makefile's variables (`CLIBSRCS, CSRCS, BINNAMES, LIBNAME`)
- Add the executable program's dependency rules

This completes the preparations for the build.

11 Common Problems

11.1 Sizes of Custom Data Structures and the SDK Library Do Not Match

RevolutionSDK libraries are built by adding the “-enum int” compiler flag. This is to force all enumerated types to use integers instead of the smallest possible data types. This flag should be added.

11.2 A “‘_main’ not found” Error Occurs During Linking

This probably indicates that a dependency list has not been prepared in the makefile for a given module where the list of objects to be linked is located. Add the correct makefile rules (refer to "9.2.2 Specifying Executable Programs to Build" on page 24).

11.3 Make Indicates ".o" Files Cannot Be Found Yet It Runs The Second Time Around

The definition for `REVOLUTION_SDK_ROOT` is probably incorrect. The `_ROOT` variable should start with `D:\RVL_SDK`. Also, note that GNU Make cannot search for directories that did not exist when Make was called. In other words, if the `obj/RVL` directory (containing “`.o`” files) did not exist when Make was called, the makefile will create it, but GNU Make will probably not be able to find any “`.o`” files that are created within that directory. In order to avoid this problem, make sure that all directories are present at all times. In order to avoid this problem, make sure that all the directories exist the whole time. In this build system, call Make two times for each module to avoid this problem. The first call will set up all the directories, and the second will perform the build.

11.4 The Make Clobber Command Hangs When Deleting bin/RVL/*

An “`.elf`” and/or project file (`.mcp`) is still open. Close them with the debugger.

11.5 The Debugger Won't Stop At the Main Function

This is happening because symbols have not been turned on during *both* the compilation of the source and linking. Run `mwldeppc` and `mwcceppc` with the `-g` option added to both.

11.6 Breakpoints Cannot Be Set In the Debugger

Use the “`-opt off`” option to turn optimization completely off.

Appendix A. Operation of commondefs and modulerules

A.1 Note

This build system is being improved each day, so it is not possible to guarantee that all information in this appendix is the latest information. This appendix focuses on the important parts of build operations (understanding how the main tree is built) and running the compiler (understanding how to use the compiler with your own build system).

A.2 commondefs Conventions

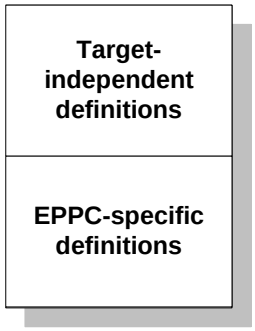
The following standard naming conventions are included.

- `CC`: C compiler
- `LD`: Linker
- `AS`: Assembler
- `AR`: Archiver (for building libraries)
- {`CC`|`LD`|`AS`|`AR`}: Flags to pass to the appropriate application

A.3 commondefs

Refer to the `/build/buildtools/commondefs` file. Figure A–1 shows the high-level structure of the file related to the targets that make it up. Note that its content does not necessarily reflect the absolute newest version, and does not necessarily accurately explain the details of this `commondefs` file. That being said, it should be sufficient information for developing your own build system.

Figure A–1 Structure of commondefs



First are the common definitions that do not depend on the target. The next entries are the most important.

Code A–1 ROOT

```
ifndef _ROOT
ROOT        = /
else
# the quotes assume Windows-style path names
ROOT        = $(subst $(space),\$(space),$(subst \,/,$(_ROOT)))
endif
```

ROOT: Specifies the location of the RevolutionSDK tree. By default, this is the `C:` drive. It can be changed by defining `REVOLUTION_SDK_ROOT` within the `RVL_DEV.bat` file. Note that GNU Make may still search in `C:\RVL_SDK` even if you change the root directory of the default RevolutionSDK tree for dependencies. If doing work on multiple trees, it is safer to make each tree have a different path.

Code A–2 ARCH_TARGET and PLATFORM

```
ifdef PLATFORM
ARCH_TARGET = $(PLATFORM)

ifeq ($(PLATFORM), HW2)
EPPC                = TRUE
MARLIN              = DI
ORCA                = TRUE
PROCESSOR           = gekko
else
ifeq ($(PLATFORM), RVL)
EPPC                = TRUE
MARLIN              = DI
ORCA                = TRUE
PROCESSOR           = gekko
BUG_TRIANGLE_FAN      = TRUE
BUG_Z_BEFORE_TEX      = TRUE
BUG_NO_8b_SCALE       = TRUE
else
:
```

ARCH_TARGET: This build system manages multiple architecture targets, so it is important to recognize which build to perform. The build system can manage an arbitrary number of targets for each unique build flag. The only valid target at the present time is the RVL target.

Code A–3 LIB_ROOT/DEMO_ROOT

```
LIB_ROOT       = $(BUILD_ROOT)/libraries
DEMO_ROOT      = $(BUILD_ROOT)/demos
```

LIB_ROOT/DEMO_ROOT: These paths are defined here so that the makefiles of local modules can identify which path to use during a build. By setting certain variables, the makefiles of these local modules can choose which path to use.

Code A–4 MWDIR

```
MWDIR   = $(subst $(space),\$(space),$(subst \,/,$(CWFOLDER)))
```

MWDIR: This specifies the location of the CodeWarrior tree that is defined as CWFolder in autoexec.bat. As a result, the path must be converted into a format that is easier to use (in other words, backslashes and spaces must be converted into their UNIX-compatible counterparts).

Code A–5 CCFLAGS

```
CCFLAGS        = -nodefaults -proc $(PROCESSOR) -D$(ARCH_TARGET) -align powerpc –W all –enum int
```

CCFLAGS: Common C compiler flags. Similar arguments also apply to assembler (ASFLAGS). Their various meanings are shown below:

- -nodefaults: Causes the compiler to check the environment variables for library paths. This flag reduces the size of the environment variables.
- -proc $(PROCESSOR): Indicates which CPU the generated code is for. For the PROCESSOR portion, "gekko" is defined for RVL builds.
- -D$(ARCH_TARGET): Defines "RVL" in the source code in order to perform a conditional compile based on the target.
- -align powerpc: Uses the PowerPC default alignment (4-byte words).
- -W all: Performs all standard (ANSI) error checks.
- -enum int: Causes enumerated types to always become complete integer types (in other words, 4 bytes). Based on the ANSI standard.

Code A–6 ARFLAGS

```
ARFLAGS         = -nodefaults -xm l
```

ARFLAGS: These are the common archive generation flags that are used for generating libraries.

- "-xm l" means "Create a library"

In the same place, add the following options depending on whether you are building a debug target or an optimized target.

Code A–7 ARFLAGS Argument Options

```
ifdef DEBUG
CCFLAGS         += -opt off -D_DEBUG -inline off
ASFLAGS         += -D_DEBUG
else
CCFLAGS         += -DNDEBUG -O4,p -inline auto
ASFLAGS         += -DNDEBUG
endif
```

-D_DEBUG/-DNDEBUG: _DEBUG and NDEBUG are the symbols that can be used within the code to identify whether or not the build is a debug build. By convention, system code uses _DEBUG exclusively for debug builds. NDEBUG is used with optimized builds to make them ANSI C compliant. _DEBUG is defined for debug builds, but is not defined for optimized builds. NDEBUG is not defined for debug builds, but *is* defined for optimized builds.

- -opt off: No optimization. The CodeWarrior debugger won't function well if any kind of optimization is done whatsoever.
- -inline off: No inline functions. Inline functions introduce some confusion into debug code.
- -O4,p: Complete optimization. This is optimization for speed (the opposite of code size).
- -inline auto: Makes all small functions into inline functions when possible and when effective, even if "inline" has not been specified.

A.4 EPPC-Specific Flags

Code A–8 MWCLDIR

```
MWCLDIR       = $(MWDIR)/PowerPC_EABI_TOOLS/Command_Line_Tools
```

MWCLDIR: Specifies the location of the command-line tools.

Code A–9 EPPCMWLIBS

```
MWLIBDIR      = $(MWDIR)/PowerPC_EABI_Support/Runtime/Lib

# H is for hardware floating point
EPPCMWLIBS    = $(MWLIBDIR)/Runtime.PPCEABI.H.a \
                $(MWDIR)/PowerPC_EABI_Support/MSL/MSL_C/PPC_EABI/Lib/MSL_C.PPCEABI.bare.H.a \
                $(MWDIR)/PowerPC_EABI_Support/MSL/MSL_C++/PPC_EABI/Lib/MSL_C++.PPCEABI.bare.H.a
```

EPPCMWLIBS: Specifies the required Freescale libraries.

Code A–10 LIBS

```
LIBS   = $(INSTALL_ROOT)/lib/base$(LIBSUFFIX) \
                $(INSTALL_ROOT)/lib/os$(LIBSUFFIX)    \
                $(INSTALL_ROOT)/lib/db$(LIBSUFFIX)    \
                $(INSTALL_ROOT)/lib/mtx$(LIBSUFFIX)   \
                $(INSTALL_ROOT)/lib/dvd$(LIBSUFFIX)   \
                $(INSTALL_ROOT)/lib/vi$(LIBSUFFIX)    \
                $(INSTALL_ROOT)/lib/demo$(LIBSUFFIX) \
                $(INSTALL_ROOT)/lib/pad$(LIBSUFFIX)   \
                $(INSTALL_ROOT)/lib/ai$(LIBSUFFIX)    \
                $(INSTALL_ROOT)/lib/ar$(LIBSUFFIX)

# GX related libraries
LIBS  += $(INSTALL_ROOT)/lib/gx$(LIBSUFFIX)    \
                $(INSTALL_ROOT)/lib/G2D$(LIBSUFFIX)          \
                $(INSTALL_ROOT)/lib/geoPalette$(LIBSUFFIX)   \
                $(INSTALL_ROOT)/lib/texPalette$(LIBSUFFIX)   \
                $(INSTALL_ROOT)/lib/fileCache$(LIBSUFFIX)    \
                $(INSTALL_ROOT)/lib/support$(LIBSUFFIX)

# Char pipeline related libraries
LIBS  += $(INSTALL_ROOT)/lib/control$(LIBSUFFIX)     \
                      $(INSTALL_ROOT)/lib/actor$(LIBSUFFIX) \
                      $(INSTALL_ROOT)/lib/anim$(LIBSUFFIX)  \
                      $(INSTALL_ROOT)/lib/lighting$(LIBSUFFIX) \
                      $(INSTALL_ROOT)/lib/shader$(LIBSUFFIX)
```

LIBS: Specifies all of the system libraries. All unreferenced code will be “dead stripped” by the linker.

Code A–11 INCLUDES

```
# -i- delimits user paths from system paths.
# By default, the only user path points to the local include tree.
# -ir means recursive descent
LINCLUDES   = -I$(MODULE_ROOT)/include
GINCLUDES   = -I$(INC_ROOT)                                  \
              -ir $(MWDIR)/PowerPC_EABI_Support/Msl/Msl_c    \
              -I$(MWDIR)/PowerPC_EABI_Support/Runtime/Inc

INCLUDES    = $(LINCLUDES) -i- $(GINCLUDES)
```

INCLUDES: Specifies the include paths that are supported by the build system. The include paths are divided into the “local” user paths and the “global” system paths. The “-i-” [notation] is the delimiter between the system paths (included by using <>, for example <.h>) and the user paths (included by using “”, for example “gxprivate.h”). Only the user paths specify the module-specific include directories. Specify “//include” for the Nintendo GameCube system path. All other paths are specific to Freescale.

Code A–12 CCFLAGS

```
CCFLAGS        += -fp hardware -Cpp_exceptions off
```

CCFLAGS: The EPPC tools must recognize whether floating-point hardware is available. Also, an additional runtime setup is required in order to use C++ exceptions, so this must be turned off.

Code A–13 LDFLAGS

```
LDFLAGS += $(LIB_PATH) -fp hardware $(EPPCMWLIBS) -unused -map ${@:.elf=.map}
```

LDFLAGS: The linker must have the same hardware floating-point settings as the remaining portion of the application. There is no reason not to make use of the hardware floating-point. The linker will also generate a “.map” file that contains the locations of all symbols. This is stored in the same location as the “.elf” file.

Note that the -g option is added to CCFLAGS, LDFLAGS, and ASFLAGS. This enables symbols for debugging for all builds.

Code A–14 CC/LD/AR/AS

```
CC            = $(MWCLDIR)/mwcceppc.exe
AS            = $(MWCLDIR)/mwasmeppc.exe
LD            = $(MWCLDIR)/mwldeppc.exe
AR            = $(MWCLDIR)/mwldeppc.exe
```

CC/LD/AR/AS: Indicate the CodeWarrior tools that correspond to the compiler, linker, archiver, and assembler, respectively.

A.5 Debug Flags

The binaries of debug versions and optimized versions not only require different linker and compiler flags, but also have different names. These are described in the following portion of the commondefs file.

Code A–15 Debug Flags for commondefs

```
ifdef DEBUG
LIBSUFFIX    = D.a
BINSUFFIX    = D.elf
DOLSUFFIX    = D.dol
else
LIBSUFFIX    = .a
BINSUFFIX    = .elf
DOLSUFFIX    = .dol
endif
```

A.6 modulerules

All GNU Make rules for deciding how to build objects are described in the `/build/buildtools/modulerules` file. The file is composed of the following parts.

- Path setup
- Generation of the list of objects to build
- Common build and linking rules
- Deletion and setup of other targets
- Installation of modules (only system libraries and demos)

A.7 Path Setup

Code A–16 Path Setup

```
ifeq ($(DEMO),TRUE)
PROJ_ROOT      = $(DEMO_ROOT)
endif

ifeq ($(LIB),TRUE)
PROJ_ROOT      = $(LIB_ROOT)
endif

ifeq ($(TEST), TRUE)
PROJ_ROOT      = $(TEST_ROOT)
endif

ifeq ($(SAMPLE), TRUE)
PROJ_ROOT      = $(SAMPLE_ROOT)
endif

ifeq ($(CHAR_PIPELINE), TRUE)
PROJ_ROOT      = $(CHAR_PIPELINE_ROOT)
endif

ifeq ($(BUILD_TOOL), TRUE)
PROJ_ROOT      = $(MAKE_ROOT)
endif
```

Each module's makefile must define which subtree it belongs to from among those defined in `DEMO`, `LIB`, `TEST`, and `SAMPLE`. `PROJ_ROOT` is used to generate all other paths used by the compiler, as shown below.

Code A–17 Secondary Path Setup

```
MODULE_ROOT             = $(PROJ_ROOT)/$(MODULENAME)
FULLSRC_ROOT    = $(MODULE_ROOT)/src

BINOBJ_ROOT     = obj/$(ARCH_TARGET)/$(BUILD_TARGET)
DEP_ROOT        = depend/$(ARCH_TARGET)/$(BUILD_TARGET)
FULLBINOBJ_ROOT = $(MODULE_ROOT)/$(BINOBJ_ROOT)
FULLDEP_ROOT    = $(MODULE_ROOT)/$(DEP_ROOT)

BINLIB_ROOT     = lib/$(ARCH_TARGET)
FULLBINLIB_ROOT = $(MODULE_ROOT)/$(BINLIB_ROOT)
FULLBIN_ROOT    = $(MODULE_ROOT)/bin/$(ARCH_TARGET)
BIN_ROOT        = bin/$(ARCH_TARGET)
```

Almost all path variables use both full (absolute) and relative paths for the following reasons:

- Absolute output paths are not desirable for the linker
- Relative paths between targets and their dependencies are not necessarily always desirable (especially if the execution of makefiles is nested)

The `*BINOBJ_ROOT` variables describe the location of the intermediate object files (`.o` files). This location will be either `obj/DEBUG` or `obj/NDEBUG`, depending on the type of build (debug or optimized).

The `*BINLIB_ROOT` variables describe the include paths for libraries. In this case, it is `lib/RVL`.

The `*BIN_ROOT` variables describe the include path for executable programs. In this case, it is `bin/RVL`.

At this time, the appropriate paths are added to the beginnings of the libraries and binaries, and the appropriate suffixes are appended to the end, thereby creating the final names.

Code A–18 Path Names for Libraries and Binaries

```
# if we're building debug versions, need to change names of built libs/bins
ifdef LIBNAME
TARGET_LIB      = $(FULLBINLIB_ROOT)/$(LIBNAME)$(LIBSUFFIX)
endif

ifdef BINNAMES
TARGET_BINS     = $(addprefix $(FULLBIN_ROOT)/,$(addsuffix $(BINSUFFIX), $(BINNAMES)))
endif
```

A.8 Building Objects

First, the absolute (full) paths are used to generate the different objects for the build, as shown in the following list.

Code A–19 Object List

```
ALLCSRCS    = $(CSRCS) $(CLIBSRCS)
ALLCPPSRCS  = $(CPPSRCS) $(CPPLIBSRCS)
ALLASMSRCS  = $(ASMSRCS) $(ASMLIBSRCS)

ifdef CSRCS
OBJECTS     += $(addprefix $(FULLBINOBJ_ROOT)/, $(CSRCS:.c=.o))
endif
ifdef CPPSRCS
OBJECTS     += $(addprefix $(FULLBINOBJ_ROOT)/, $(CPPSRCS:.cpp=.o))
endif
ifdef ASMSRCS
OBJECTS     += $(addprefix $(FULLBINOBJ_ROOT)/, $(ASMSRCS:.s=.o))
endif
ifdef CLIBSRCS
LIBOBJECTS  += $(addprefix $(FULLBINOBJ_ROOT)/, $(CLIBSRCS:.c=.o))
endif
ifdef CPPLIBSRCS
LIBOBJECTS  += $(addprefix $(FULLBINOBJ_ROOT)/, $(CPPLIBSRCS:.cpp=.o))
endif
ifdef ASMLIBSRCS
LIBOBJECTS  += $(addprefix $(FULLBINOBJ_ROOT)/, $(ASMLIBSRCS:.s=.o))
endif

ALLOBJECTS  = $(OBJECTS) $(LIBOBJECTS)
```

With this, it is possible to generate simple GNU Make rules that include all types of objects.

Code A–20 GNU Make Rules for Building Objects

```
# GENERIC C FILE BUILDING
$(FULLBINOBJ_ROOT)/%.o: src/%.c
        @if [ ! -d $(@D) ] ; then \
                echo ">>> Creating $(@D)" ; \
                mkdir -p $(@D) ; \
        fi
        @if [ ! -d $(dir $(FULLDEP_ROOT)/$*) ] ; then \
                echo ">>> Creating $(dir $(FULLDEP_ROOT)/$*)" ; \
                mkdir -p $(dir $(FULLDEP_ROOT)/$*) ; \
        fi
        @if [ ! -d $(MODULE_ROOT)/include ] ; then \
            echo ">>> Creating $(MODULE_ROOT)/include" ;\
                mkdir -p $(MODULE_ROOT)/include ; \
        fi
        @echo
        @echo ">> $< --> $(BINOBJ_ROOT)/$(subst $(FULLBINOBJ_ROOT)/,,$@)"
ifdef DISASSEMBLE
        $(CC) $(CCFLAGS) $(INCLUDES) -dis $< > $<.s
endif
ifdef EPPC
        $(CC) $(CCFLAGS) $(INCLUDES) $(COMPILE) $< \
                -o $(subst $(FULLBINOBJ_ROOT)/,$(BINOBJ_ROOT)/,$@) -MD
        @echo -n "$(FULLBINOBJ_ROOT)/$*.o: " > depend.tmp ; \
        cat $(notdir $*).d | sed -f "$(BUILDTOOLS_ROOT)/eppccleandepend.sed" >> depend3.tmp ;\
        cat depend3.tmp >> depend.tmp ; \
        mv depend.tmp $(FULLDEP_ROOT)/$*.d ; \
        rm -f depend*.tmp *.d
else...
```

`$(FULLBINOBJ_ROOT)/%.o: src/%.c`: This target matches the `.o` files in `$(ALLOBJECTS)`.

First, this rule verifies that the output directories exist.

Compile: The actual compile line uses the `$(COMPILE)` option of `-c` for EPPC. Note that with this rule, “relative” paths are used instead of “full” output paths. With absolute output paths, the compiler will simply exit.

A.9 Generating Dependencies

File dependencies are generated at the same time as the compilation.

First, a new rule target is generated from the rules in Code 26 and output to `depend.tmp`. This target contains the full paths of the object files. Ultimately, this file is added before compiler output.

The compiler generates the dependencies and object code when given the "`-MD`" option, and this output is saved as "`$*.d`" (in other words, the extension `.d` is added to the object file name). The compiler output is then removed using a sed script (`/build/buildtools/eppccleandepend.sed`), and output to `depend3.tmp`.

Finally, `depend.tmp` is added to `depend3.tmp`, and the clean rules are generated.

The sed script contains the following content:

Code A–21 sed Script

```
# change all back slashes
sX\\X/Xg

# eliminate quotes
sX\"XXg

# change all spaces
sX/\ X\\ Xg

# except the separateor after colon
sX:\\X: Xg

# replace last slash with backslash
sX/$X \\Xg

# kill everything before the colon
sX.*o:XXg
```

The following are problems that can be caused when outputting dependencies from the compiler:

- Using a backslash instead of a forward slash
- Enclosing paths in quotes
- Including spaces in paths (spaces must be preceded with a backslash)
- Target paths don’t work with this build system

A.10 Object Header Rules

The latter half of the `modulerules` file contains the targets for all objects to be built, as shown below.

Code A–22 Build Targets

```
dobuild:$(OBJECTS) $(TARGET_LIB) $(TARGET_BINS)
```

If you recall, the build targets are specified in a module's makefile. In other words, each makefile designates what must be performed during `setup` and `build`. In practice, the two targets `build` and `dobuild` are used. The build target does not include dependency files, so these files won't necessarily always be generated during a `clobber` or `clean`. Note that with modules, the normal Make command will run the `build` target first in order to configure all the directories. After that, Make is run once again for the `dobuild` target. This not only avoids the inclusion of dependency files for `clobber` and `clean`, but also avoids a `VPATH` bug with GNU Make.

A.11 Linking

Code A–23 Linking

```
# GENERIC .elf/executable BUILDING
# linker command file is used whenever LCF_FILE is defined.
# Note it is only used when linking executables.  LCFs are only useful
# for EPPC.
ifdef LCF_FILE
LDFLAGS        += -lcf $(LCF_FILE)
endif

$(FULLBIN_ROOT)/%$(BINSUFFIX):
        @echo
        @echo ">> $(notdir $+) --> $(subst $(FULLBIN_ROOT)/,,$@)"
        $(LD) $(LDFLAGS) $+ -o $@
```

Once the flags have been set up, linking can be done easily.

`LCF_FILE`: The linker command file is used by the EPPC target for the layout of sections within the executable program. Note that LCF files are automatically generated in the current build system.

`$(LD) $(LDFLAGS) $+ -o $@`: This simply creates a target (`$@`) using all objects specified in the dependency rules (`$+`) of the module's makefile. Do not forget that additional Freescale libraries are included with `$(LDFLAGS)`.

Libraries use similar rules like the following.

Code A–24 Library Linking

```
# GENERIC LIBRARY LINKING
$(TARGET_LIB): $(LIBOBJECTS)
        @echo
        @echo ">> New $(notdir $?) --> linking $@"
        $(AR) $(ARFLAGS) -o $(TARGET_LIB) $(LIBOBJECTS)
```

Appendix B. How to Boot NDEV

This appendix outlines what happens when `ndrun` is entered at a bash prompt or when a debugging session is started. It should help you out with any problems that you may encounter during setup.

The `ndrun` script performs the following operations:

1. Converts `.elf` executable programs into a format that the OS can boot (with emulated DVD drives, this format can be seen as `default.dol`). Details about this format can be seen in `/include/revolution/dolformat.h`. There are trivial limitations on both the text and the number of static data segments.
2. Flushes the Windows disc cache. Note that a problems may occur with Windows becoming slow if emulated DVD drives are being shared on a network.
3. Generates several data structures that describe the layout of the emulated DVD disc.
4. Notifies the DVD drive emulator that the disc has changed.
5. Reboots NDEV.

The compiler uses the `ndrun` script in order to start debugging sessions. Furthermore, it sets a breakpoint in the `main` function that starts the debugging session.