Coyotos Developer Guide

Version 0.1

Jonathan S. Shapiro, Ph.D.
The EROS Group, LLC

September 10, 2007

Contents

Chapter 1

Introduction

This book will show you how to get started with Coyotos. They setting up your development machine, obtaining the source code, where to get help, the tutorials, Coyotos "theory of operation", and so forth.

There are really two types of Coyotos developers:

- Embedded system developers who want to know how to assemble a complete Coyotos system, including kernel, drivers, and application structure. These developers are interested in the "system view" of Coyotos development.

- Application or driver developers who have an existing Coyotos system running and want to develop a new application or utility. These developers generally will not be building a system from the ground up. Instead, they are developing applications that will be installed by the Coyotos installer.

At the moment, the information in this book is biased in favor of embedded system developers, but a lot of that information is relevant for application developers as well. The squeaky customer gets the grease and all that.

Chapter 2

Setting Up Your Development Machine

The Coyotos development environment is currently supported on Fedora Core 8 and later. Since they are obsolete, we will be dropping support for Fedora 8 through 10 shortly. We also build for CentOS 5.4. The CentOS packages *should* work on RHEL5.4 as well, but we do not test that. Some users are using other versions of Linux or BSD, but this involves some extra work and we do not actively support it. This page describes how to set yourself up on Fedora Core.

If you want to be notified when the source base is updated, you should also subscribe to the coyotos-commit notification list at http://www.eros-os.com/mailman/listinfo.

2.1 Installing the Cross Compilers

The first thing to do is to install the RPM package that adds the Coyotos tool repository to your environment. To do this, you should (as root):

```
rpm -ivh http://www.eros-os.com/YUM/coyotos/Fedora/12/Coyotos-Repository.noarch.rpm
```

If you are running an earlier version of Fedora Core, replace the "12" accordingly. If you are running CentOS 5.4, use:

```
rpm -ivh http://www.eros-os.com/YUM/coyotos/CentOS/5/Coyotos-Repository.noarch.rpm
```

Once this package is installed, you will be able to install the Coyotos cross tools. We currently provide cross compilation kits for IA-32, Coldfire, and ARM/StrongArm targets [the ARM tools are not tested by us, but are actively used by the CapROS team. Depending on your target of interest, you should run one or more of:

```
yum install coyotos-i386-xenv
yum install coyotos-m68k-xenv
yum install coyotos-arm-xenv
```

That may take a while. Be patient. The advantage to installing this way is that you will automatically receive updates. Occasionally we have been known to make changes to the source tree that rely on an update in the cross tools. This is happening less and less often over time, but it still happens once in a while.

If we don't supply packaged tools for your platform, the altermative is to build them yourself. Instructions for this will be moving here shortly. For now, they can be found at the old site.

2.2   Check Out the Source Code

If you don't already have it, you will need to install `mercurial`. We use mercurial for our source code management. You can find information abuot how to use mercurial at http://selenic.com/mercurial/wiki. There is a pre-packaged version of mercurial in the Fedora repository for all recent versions of Fedora. While you are installing mercurial, you may also find the `colordiff` package useful.

2.2.1   Set Up Mercurial

The Coyotos makefiles rely on the `mercurial fetch` extension. You may also want to enable the `gpg` extension. Both extensions ship with `mercurial` out of the box. To enable these, create a file `.hgrc` under your home directory that enables the extention. Here is an example of a minimal `.hgrc` file:

```
[ui]
username=Danny Developer <danny@coderworks.org>
editor=vi

[extensions]
hgext.fetch=
hgext.gpg=
```

2.2.2   Getting the Code

Checking out the base Coyotos tree can be done with the following command. You may wish to consider doing this within an empty containing directory:

```
hg clone http://dev.eros-os.com/hg/coyotos/trunk coyotos
```

If you wish, you may also want to consider checking out the tutorial subtree:

```
cd coyotos/src
hg clone http://dev.eros-os.com/hg/tutorial/trunk tutorial
```

The other package that is currently available is the **ccs-xenv** package. It should also be cloned under **coyotos/src** if you choose to check it out. You don't actually need it unless you are actively working on the cross tools (e.g. you are porting to a new target).

With a bit of luck, you should now be able to build the tree with:

```
cd coyotos/src
make
```

Chapter 3

Tree Organization, Building, and Rebuilding

The Coyotos tree is divided loosely into "packages" (which is a terrible name). In general, the rule is that if a directory `mumble/` appears under `coyotos/src`, it is a self-contained "unit" of the build. The exception to this statement is the `coyotos/src/build/` directory, which contains makefile support, scripts for installing binaries, and so forth.

The build order and dependencies for the current subdirectories of `coyotos/src` are:

Order	Tree	Dependencies	Description
1	`ccs-xenv/`	*none*	This tree produces the cross compilers. It is not built automatically. If you are using Fedora as your development environment, you will probably never build this tree because you have installed the cross environment packages from our repository.
2	`ccs/`	`ccs-xenv/`	This tree produces additional build tools that are not yet ready to move into packaging. Logically, `ccs/` and `ccs-xenv/` form a single set of tools. This tree produces the `capidl` and `mkimage` tools, which are needed to generate interfaces, header files, IPC stub libraries, server demultiplexing stubs, and complete system images.
3	`sys/`	`ccs/`	Kernel source code. Relies on `capidl` to generate interfaces and opcode macros.
4	`base/`	`sys/,ccs/`	Core Coyotos utilities and drivers.
5	`web/`	`ccs/`	Documentation tree. Relies on the *presence* of `sys/` to obtain the kernel interface IDL specifications, but does not rely on `sys/` being compiled.
6	`tutorial/`	`base/`	Tutorial segments for various aspects of Coyotos. These rely on the utilities and drivers generated from `base/`. The tutorial/ tree is an optional tree. If it is not present, we will not try to build it.
7	`var.`*`something`*	`base/`	Any subdirectory of `coyotos/src` whose name is `var.`*`something`* is assumed to contain code added by you. If one or more directories of these names are present, they will be built in alphabetical order.

It is assumed that any subdirectory named `var.`*`something`* conform to the Coyotos build system conventions. If these subtrees are maintained using mercurial, our top-level makefile (the one in `coyotos/src`) can be used to help keep them up to date and to see their status. The addition of new subdirectories necessarily will drag you into the black art of Makefile management, which is documented later in the *Developer Guide*.

5

Output directories `coyotos/host` and `coyotos/usr` are created by the build process. The `coyotos/host` directory is where tools used on the development host (from `ccs/`) are installed. The `coyotos/usr` directory is where all of the build results are installed. The general pattern of the build is that each package is built in sequence and installs its results into `coyotos/usr` and/or `coyotos/host` as appropriate. In *theory*, it should be possible to build each subtree in its proper order, delete that subtree, and then build the next subtree. In practice, this would probably break a number of parts of the build.

The `web/` subtree is something of an unfortunate special case. It is automatically rebuilt to produce pieces of the Coyotos web site. Some of the documents rely on the IDL files of other trees in order to generate interface documentation. At the moment, these IDL files are extracted from their positions in the source tree rather than from their installed locations. This is a bug that has not yet been addressed.

3.1   Re-Building the Tree From Scratch

We are currently (perhaps I should say constantly) working to improve the build process, especially in the documentation subtree. As with any operating system build tree, there are build order dependencies from one part of the tree to the next. Not all of these are capturable in the Makefiles. The big issue is that later subtrees depend on interfaces that are published by earlier subtrees.

Because of this, the first thing you should do if you suspect you have a build problem is to rebuild the tree from scratch. The simplest way to do this is:

```
cd coyotos/src
make pristine
```

This is a short-hand for the following commands:

```
cd coyotos/src
make distclean    # clean out the tree and everything that has been installed
make              # 2m37s elapsed with cold ccache, 2m16s with warm ccache
```

The Coyotos build is designed to use `ccache` if you have it installed. The `ccache` tool does amazing things to reduce build times, and we strongly recommend it. Ironically, this leaves the documentation tree as the "expensive" part of the build. If you are in a hurry, there are a couple of things that you can do to reduce build times.

- You can skip the build of the documentation tree entirely. Setting OPTIONAL_DIRS="" on your make command line will skip the build of the doc/ and tutorial/ subtrees.

  ```
  make OPTIONAL_DIRS=""  # 21s elapsed (warm ccache) # 45s
  elapsed (cold ccache)
  ```

- You can suppress generation of PDF files by restricting the targets for OSDoc. It is the PDF files that take all of the time:

  ```
  make OSDOC_TARGETS="%.html"  # 30s elapsed (warm ccache)
  ```

Please note that you should *never* change the value of OPTIONAL_DIRS when you are running `make clean` or `make distclean`. If you do so, those subdirectories will not be visited by the cleaning process. As a result, you will not get a clean build later.

3.2   Rebuilding A Particular Subtree

The convention in the Coyotos tree is that typing `make package` from within a package subtree rebuilds that subtree (from the top of the subtree) and installs its output. This build process assumes (without checking) that previous

subtrees have been built. It also assumes that the results of previous builds of the current subtree will not alter the build result. This creates a requirement of convention: *where the build of a given package requires interface specifications or mkimage component files that are provided by the same package, it should obtain these from the package source directory, not from the installed location.* You can also type `make world` or `make pristine`, either of which will rebuild the entire tree proceeding from `coyotos/src`. If you simply type `make`, what happens depends on your current directory. If you are sitting in `coyotos/src`, typing `make` is equivalent to typing `make world`. If you are sitting within a package subtree, typing `make` is equivalent to typing `make package`.

At this point, we need to explain one piece of the makefile magic: how do the makefiles figure out where you are in the subtree? In principle, you could check out your tree anywhere. Two rules are applied in sequence:

1. If an environment variable **COYOTOS_ROOT** is set, it will be used. **COYOTOS_ROOT** tells the build system where the top of your tree is. It is not required that the top-level directory be named `coyotos/`, but it will avoid confusion if you follow this convention. The tree *underneath* the ${COYOTOS_ROOT} directory must follow the coyotos conventions.

2. If **COYOTOS_ROOT** is not set, the build system will examine your current working directory to see if it is of the form *some/path*/`coyotos/src/` or *some/path*/`coyotos/src/`*subdirectory*. If so, it will assume that *some/path*/`coyotos/src/` is the top of your development tree. As a practical matter, I don't think that any of the Coyotos or EROS developers (EROS makefiles used a similar trick) have bothered to set **COYOTOS_ROOT** in the last decade.

A minor word of caution: the mechanism used to find *some/path*/`coyotos/src/` is not sophisticated. If you have a directory of the form *path*/`coyotos/src/`*more/path*/`coyotos/src`, please let us know what it does.

Chapter 4

What a Coyotos System Looks Like

Viewed as a whole, a Coyotos system consists of some number of running programs that are "connected" by capabilities. The result is a graph, in which entry capabilities indicate dependencies. If my program relies on yours to provide some function, then my program will call yours. In order to do that, my program will hold an entry capability to your program. If we were to freeze the system, and draw a graph showing all of the entry capabilities *other than* brand capabilities (see below), we would have a picture illustrating the runtime software dependency graph.

This graph is dynamic. New processes are created. They in turn create new helper processes. Later, these processes finish, tearing down some or all of the structure that they have built. The dependencies that we are looking at are dynamic, run-time dependencies. In a well-structured system, we would find that there are no cycles in this graph. A cycle would indicate a path where some program might deadlock by depending on itself recursively.

In contrast to most other systems, Coyotos processes are not built from files. Actually, it is more the other way around: the "file" object is really an interface implemented by one (or more) processes. In fact, the general pattern in Coyotos is that every process implements one or more objects.

4.1   Interfaces, Objects, and Permissions