

The Coyotos Build System[†]

Version 0.3

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

Abstract

Coyotos requires a regrettably complicated build procedure. The complexity is mainly driven by two issues: the need for cross compilation and the fact that some of the things in the build tree come from other (foreign) build environments — most notably things built with autoconf.

This document describes how to obtain and install the cross tools, how the build tree is structured, what the complications are, and how the Coyotos build mechanism is intended to operate.

The description provided here is provisional. We are testing a new, simplified build process relative to the older EROS build process.

Contents

1	Setting up a Build Environment	2
1.1	Recognized Targets	2
1.2	Setting up on Coyotos	2
1.3	Setting up on Linux/YUM Systems	2
1.4	Setting up on Other UNIX's	4
1.5	Setting up on Windows	4
2	Obtaining the Sources	4
3	Organization of the Source Tree	5
3.1	Impact of IDL	6
3.2	Normal Build Sequence	6
4	Adding a New Package To The Tree	7
5	Coyotos Makefiles	9
5.1	XML Documentation	9
5.2	Building Object Code	10

[†] Copyright © 2005, Johns Hopkins University.

1 Setting up a Build Environment

The Coyotos development team runs Fedora. At the time of this writing, we use Fedora 7, but our usual practice is to run the latest Fedora release. In practical terms, this means that if Fedora has updated recently you can probably still rely on the cross environment working, but you should probably upgrade soon, because we will only be working on (and therefore testing) the latest release.

We try to tinker with the cross environment as little as possible, so it is fairly pointless to rebuild it every time the source tree is built. On the other hand, it *does* need to be updated once in a while, either to supply a bug fix or to add a new component. Our solution is that we have packaged the cross tools as RPM files, and we make these available via a YUM repository. **If it is feasible for you to use this YUM repository, we strongly recommend it.**

The following discussion assumes that you are building for the recent Pentium family (i386 target). We will shortly support at least one other architecture. If so, you should substitute the appropriate target name for `i386` wherever it appears below.

1.1 Recognized Targets

The targets that are currently recognized take the form “*os-architecture*” where “os” is one of “coyotos” or “capros”,¹ and “architecture” is one of:

i386 IA32 processors and derivatives, eventually including Geode and similar embedded processors. Current work is targeted exclusively at PC-based desktop machines, which is our primary development system.

It would have been better to call this family “IA32,” but the use of “i386” as a family identifier is nearly universal among `configure` scripts, and changing the label wasn’t important enough to justify the patch effort required.

m68k Coldfire processors.

arm ARM-based processors. No work for this target is currently being done in Coyotos. The target is maintained in the cross tools chain so that the CapROS project and the Coyotos project can work from a common tool chain to avoid duplication of effort.

We would eventually like to see support for **x86_64**, but no port of `newlib` has been done to that platform yet.

1.2 Setting up on Coyotos

The best development environment for Coyotos is, of course, Coyotos itself. If you cannot find a Coyotos port for your platform, *please write one*.

Okay, okay. This isn’t likely for a while, so let’s move on to those *other* systems that you may need to suffer with until an upgrade to Coyotos becomes available...

1.3 Setting up on Linux/YUM Systems

To make getting going on Linux easy, we have built a package to update your yum configuration. Once this is installed the rest is easy. To install this package:

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

If you are running an earlier version of Fedora, replace the “12” accordingly. Be advised that we will be dropping support for obsolete versions of Fedora very shortly. If you are running CentOS 5.4, use:

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

¹ We are currently maintaining a cross-build suite for the capros project.

The CentOS packages *should* work for RHEL5.4 as well, but we haven't tested that.

Once the repository is set up, you can install some helpful tools (and a required library) by running (as root):

```
yum install coyotos-i386-xenv
```

or possibly `coyotos-arm-xenv` or `coyotos-m68k-xenv`. This will get you all of the tools in the Coyotos cross environment for your target.

If you are asked whether to import the signing key, you will need to say "yes". This key is used to sign all of our packages. If you wish to ensure that the key is authentic, you can check the key fingerprint at `pgp.mit.edu` key server by searching for `packager@eros-os.org` if you choose to do so. If imported, you should find that it is signed using my (Jonathan Shapiro's) offline signing key. The fingerprint of my offline signing key is `DFAB6639`. If you export this key, you can directly import the ascii armored file using `rpm --import`.

The `coyotos-i386-xenv` package is a "virtual" package — it simply supplies the necessary dependencies so that YUM will find all of the pieces you need. As of this writing, the packages installed are:

coyotos-common-filesystem Architecture independent file system skeleton for `/coyotos`

coyotos-common-binutils Architecture independent support files for the binary utilities.

coyotos-common-gcc3 Architecture independent support files for GCC.

coyotos-common-gcc3-cpp Architecture independent support files for CPP.

coyotos-i386-filesystem Architecture specific file system skeleton for `/coyotos`

coyotos-i386-binutils Assembler, linker, and so forth for cross compilation.

coyotos-i386-newlib The `newlib` C library for Coyotos native applications. This is likely to be updated a fair bit.

coyotos-i386-gcc3-gcc GCC version 3 that targets the Coyotos target environment.

coyotos-i386-gcc3-cpp C preprocessor.

If you are targeting the Pentium family, you may also find it helpful to install `qemu`, a full-machine emulator for the PC. We rely on this heavily for debugging, and if you don't have a lab full of machines it is probably the best way to experiment. You can install this by:

```
yum install qemu
```

The `qemu` package is a native tool, so it installs in the customary directories.

All of the Coyotos cross environment tools install into `/coyotos/`. If you install them with YUM, you can remove them by:

```
rpm -e `rpm -q -a |grep coyotos`  
rpm -e qemu libsherpa libsherpa-devel
```

If you are on a RPM system but you prefer *not* to use the YUM-based method, you are certainly welcome to install these packages by hand, but it will be harder to stay up to date. Alternatively, you can build the cross environment yourself (see below).

We would be willing to set up the necessary metadata for users of `apt` as well, but we would need some guidance. A further concern is that we don't tend to run and Debian systems, so it would be difficult to test the RPMs for that environment.

1.4 Setting up on Other UNIX's

It's really a much better plan to use the provided cross environment packages, because that way we can easily provide updates to the cross tools that will arrive in the normal way for your system. If you are building on some other UNIX derivative that has not accepted the Dao of YUM, you may find that nothing will do but to build the cross environment for yourself. If that is the case, you can fetch it from the repository using mercurial.

First, as root, you should create the directory `/coyotos` and use the `chown` command to make it writable to you and owned by you:

```
# as root:
mkdir /coyotos
chown yourlogin:yourgroup /coyotos
```

1.4.1 If You Have Constrained Space

If your space is not greatly constrained, skip ahead to "Obtaining the Sources". Fetch the entire tree and then build the cross environment from inside `coyotos/src/ccs-xenv`.

Find an empty directory with an unreasonably large amount of available disk space, `cd` to it, and proceed as follows:

```
hg clone http://dev.eros-os.com/hg/ccs-xenv/trunk ccs-xenv
(cd ccs-xenv; make -f Makefile.xenv TARGETS="coyotos-i386")
```

This will compile and install the complete cross environment. Once this is done, you can delete the cross environment source directory. It takes up a lot of space. If you need to compile the tools for multiple targets, you can list them in the `TARGETS` variable, for example:

```
(cd ccs-xenv; make -f Makefile.xenv TARGETS="coyotos-m68k coyotos-i386")
```

If you are unable to install the cross tools in `/coyotos`, you can specify another location to the `make` command:

```
(cd ccs-xenv; make -f Makefile.xenv TARGETS="coyotos-i386" CROSSROOT=$HOME/coyotos-xen
```

This will require some fiddling in the rest of the build environment, and it is not a configuration that we test. If you choose to put your tools someplace unusual, then at build time you will need to set the environment variable `COYOTOS_XENV` to point to wherever you put it (the same value that you gave for `CROSSROOT` when you built the cross environment in the command above).

Once the cross environment is installed, you can safely delete the source directory. We aren't yet rebuilding this stuff to run natively, and it consumes a very large amount of space.

1.5 Setting up on Windows

You can download a set of Fedora installation CD's from our Fedora mirror repository. When you get to the place where it asks whether you want to delete all existing partitions, remember that as much as you may *want* to, you probably can't afford to lose your Windows install quite yet.

2 Obtaining the Sources

The `coyotos` sources can be built in just about any convenient place. The top level directory of the Coyotos source tree *must* be named `coyotos`, because the makefiles rely on this to find other parts of the tree. However, the `coyotos` directory can live *underneath* any convenient location. In the following instructions, we will assume that you are placing it underneath your home directory.

The easiest way to get the sources is to use mercurial:

```
hg clone http://dev.eros-os.com/hg/coyotos/trunk coyotos
(cd coyotos/src;make get_trees)
```

Note that this should *not* be the same `coyotos` directory that you compiled the cross compilation tools into. The idea is that the directory with the cross compilation tools can be shared by multiple users. This one is for your personal build of the Coyotos tree. Placing the two trees in the same directory *will not work* (which is something we should fix at some point).

It is likely that you do not want to keep a local copy of the cross environment subtree unless you need to build it for some reason. Once the cross environment is built you can delete the `coyotos/src/ccs-xenv` subtree. The rest of the build does not rely on this subtree after the binary cross environment tools are built and installed.

3 Organization of the Source Tree

The Coyotos source tree is organized into **packages**. A package is a collection of code that is built as a unit. The build structure assumes that the Coyotos source tree is set up as follows:

```
coyotos
  src
    build      - makefile fragments for the build system
    ccs        - coyotos compilation system tools package
    ccs-xenv   - the cross environment package
    web        - the documentation tree (this web site)
    sys        - the coyotos kernel package
    base       - basic domains and libraries package
    tutorial   - package of tutorials for people getting started.
```

Each package has a top-level makefile supporting a number of standard make targets. The procedure for building a given package is determined by the package author. In practice, there seem to be four “categories” of package:

1. The *build* package, which contains the various makefile fragments and rules that support the rest of the EROS build system.
2. The *ccs* package, which contains the tools needed in order to cross-build the kernel and applications. The CCS package is a relatively large and static package, and we are working on ways to establish RPM files that will support binary-only installs of these tools.
3. So-called *native* packages, which use the Coyotos build system explained in this document.
4. *Foreign* packages — typically ones that are built from other source bases — are packages that do *not* follow the Coyotos build practices. In these cases, the package-level makefile (and the package directory) are simply wrapper makefiles that in turn invoke the foreign make process in whatever way is appropriate.

There are ordering dependencies among packages. At the moment, these are captured implicitly in a variable in the top level makefile (packages are built in the order of specification). We need to revise this so that each package can state explicitly what other packages must be built first.

The build tree is designed to sit under a directory named `coyotos`, which may live at any convenient location. This directory is named by the environment variable `COYOTOS_ROOT`. If the `COYOTOS_ROOT` environment variable is not set, the build system will attempt to locate it by searching upwards in the current working directory string to find a directory of the form `prefix/coyotos/src`. It will then assume that `COYOTOS_ROOT` should be set to `prefix/coyotos`.² From this point on in this document, when we use a path starting with `coyotos/`, it should be understood to refer to the directory named by `COYOTOS_ROOT`.

² One problem arising from this design is that confusion can result if someone creates an internal directory within the build tree named `src`. This is fairly common in UNIX-based library software, so we will undoubtedly improve this search mechanism in short order. The goal is to ensure that it is almost never necessary to set the `COYOTOS_ROOT` environment variable by hand.

3.1 Impact of IDL

Coyotos is an IDL-based system. The interfaces for Coyotos subsystems are specified using an **interface description language** (IDL). Interface description filenames end with **.capidl** for “capability IDL.” The IDL files are used to generate:

- Client-side header files needed by a client when calling a provider.
- Client-side stub libraries that translate provider invocation arguments into a Coyotos **interprocess communication** (IPC) message.
- XML documentation that describes the interface. This documentation is later gathered to create the system object reference manual.
- Server-side header files needed by a provider to *implement* an CapIDL interface.
- Server-side stub code that demarshals requests and marshals responses. This is also used to implement a CapIDL interface.

It is possible to have circular dependencies among interfaces, but not among interface implementations.

In general, the client-side IDL headers need to be generated before a given piece of client code can be compiled, and the client-side stubs need to be generated before client code can be linked. There are two ways in which this might be implemented:

1. Each client application can generate and compile it’s own copy of the necessary client headers and client stub libraries. This is feasible, but it is a nuisance and it makes the compile process take an unreasonably long time.
2. A pre-pass can be made to generate the headers and client stubs, which can then be used in common by all of the clients that depend on them. This has the advantage of not building the headers and stubs redundantly, but has the cost that the unit of build is now the package rather than the directory.

We initially tried the first approach in the EROS build process. We found that it was excessively inconvenient, and we switched to the second method. The Coyotos build process uses the second method. In the EROS build system, we found that the automatic dependency generation mechanism does a pretty good job of avoiding unnecessary rebuilds.

However, this approach creates a quandry for foreign packages, because we aren’t in a position to modify the makefiles of those packages to force a complete package rebuild whenever `make` is invoked. Even if we could resolve this, the foreign packages aren’t necessarily designed to be friendly to the complete package build discipline. Therefore:

Caution

The developer must be aware of whether they are performing a `make` within a Coyotos package or a foreign package. Within a Coyotos package, the default result of the `make` command will be to rebuild the current package. Within a foreign package, the default result of the `make` command will be whatever procedure is defined as normal by that package. Since the `makefile` at the package level conforms to the Coyotos build procedures, a `make` in the package-level directory can be relied on to rebuild the complete package.

3.2 Normal Build Sequence

When the `make` command is invoked from the `coyotos/src` directory, each package root directory is visited in order, and the `make install` command is issued for that package. The build order is:

1. The `coyotos/src/ccs` package if present,³

³ We intend to make the output of the `coyotos/src/ccs` package build available as installable RPM files for developer convenience.

2. The `coyotos/src/sys` package,
3. Packages in `coyotos/src/lib`, in the order specified in the top-level library makefile,
4. Packages in `coyotos/src/bin` in the order specified in the top-level binaries makefile,
5. Packages in `coyotos/src/examples` in the order specified in the top-level binaries makefile,

Because packages can be foreign, the internal build rules for a given package are inherently package dependent. All package-level makefiles are expected to implement the following make targets:

- `make install` Build and install all content in subdirectories of this one, then build and install the content of the current directory.
- `make world` Rebuild the entire source tree from the top.
- `make package` Rebuild the current package by issuing `make install` from the top of the package tree. In Coyotos packages, this is usually the default make target.
- `make clean` Delete all generated content in the current directory and its subdirectories.
- `make nodepend` Discard all automatically generated dependency information from the tree. Dependency information is usually updated automatically by the Coyotos makefile rules, but it sometimes becomes out of date and must be forcibly deleted. When doing this, it is often necessary to use `make -k`, and it is recommended that a clean should be performed as well:

```
make -k clean nodepend
```

Each of the make targets above is also available at the `coyotos/src` level of the tree, but the scope of effect is the entire source tree rather than a single package. In addition, the following targets are available at the top level:

- `make targdir-clobber` Delete the entire installed directory tree.
- `make pristine` Wipes out the installed directory tree, performs a clean on every package, removes all automatically generated dependencies, and then rebuilds the entire tree from scratch.

At the top level of the source tree, the default make target is `make world`.

4 Adding a New Package To The Tree

Adding a new package to the Coyotos build tree is actually fairly simple:

1. Create a new directory below `coyotos/src/lib`, `coyotos/src/bin`, or `coyotos/src/examples`.
2. Place an appropriate package-level makefile in that directory, according to whether you are implementing a foreign package or a Coyotos package.
3. Append your package directory to the **DIRS** variable in `coyotos/src/lib/Makefile`, `coyotos/src/bin/Makefile`, or `coyotos/src/examples/Makefile`, as appropriate.

Note that we will adding package-level dependency tracking in the future, which will hopefully allow you to rebuild only those packages that are needed to support the package you are trying to test. As the source tree grows, we expect that this capability will become increasingly attractive.

Here is an annotated example makefile for the package level:

```

# Copyright notice

# Default target. Every makefile should specify a default
# target. This should be the first non-comment content in
# the makefile. Package-level makefiles should specify
# package as their default target.

default: package

# Relative path to coyotos/src. All
# makefiles rely on being able to locate the top-level source
# directory in order to include other makefile fragments.
# The definition of COYOTOS_SRC should appear immediately
# beneath the identification of the default make target.
COYOTOS_SRC=../..

# Subdirectories to build. This is a list of directories
# that should be visited in sequence by the normal build
# process. The Coyotos makefiles use the DIRS variable
# to drive build recursion. Therefore, foreign packages
# should NOT specify subdirectories using the DIRS variable!
DIRS=subdir1 subdir2 subdir3

# Additional directories to clean [optional]. This is useful
# if you have directories such as test directories that are
# designed to be built by hand. These should not be visited
# by the normal build mechanism, but you probably want them
# cleaned by make~clean. Every element of DIRS will
# automatically be added to CLEANDIRS.
CLEANDIRS=otherdir

# Include the generic package-level makefile support stuff.
# This defines all of the package-level make targets:
include $(COYOTOS_SRC)/build/make/pkgrules.mk

# COYOTOS PACKAGES:
#
# That's it. The real work is done by the included makefile
# fragments by recursing using the DIRS variable.

# FOREIGN PACKAGES ONLY:
#
# Additional targets (if any) that you may want should be
# defined here. Foreign packages will want to define
# targets to execute the build and to perform a recursive
# clean:
local-install:
    # Commands to (recursively) build and install
local-clean:
    # Commands to (recursively) clean

# Add dependencies so that the clean and install targets
# will do the right thing:
install: local-install

```

```
clean: clean-install
```

5 Coyotos Makefiles

If you are trying to build a wrapper package for a foreign code base, you are largely on your own. The package-level makefiles supply the `make world`, `make package`, `make nodepend`, `make install`, and `make clean` rules. Filling in the details is up to you. The balance of this document focuses on makefiles that are built to follow the Coyotos build conventions.

The package level makefile is really just a directory wrapper that directs recursion. Nearly the same makefile structure is used in each container directory within the Coyotos source tree. The only difference is that Coyotos directory makefiles include `makerules.mk` rather than `pkgrules.mk`. Eventually, the recursive build procedure descends into some directory where real work needs to be done. At that point, it becomes necessary to create a more complicated makefile, and we need to start explaining which make variables do what.

The automatic build rules supported by the Coyotos build system support generating documentation from XML, generating code from C/C++, and generating code from BitC. All of these perform automatic dependency generation where appropriate.

The general pattern of a Coyotos makefile is:

- Define default target and path to top of source tree
- Define framing makefile variables
- Include `makerules.mk`
- Add target-specific rules, if any.

Each type of generated output has its own set of framing makefile variables, which are described below.

5.1 XML Documentation

The Coyotos documentation system is based on the assumption that Coyotos documentation is written in XML using the `osdoc` DTD. The Coyotos documentation tree includes a set of XSLT transformers that produce both online (HTML) and offline (PDF) versions of these documents. The PDF versions are generated using LaTeX.

The document generation process is controlled by the `XMLSOURCE` variable. This variable should be defined *before* including `makerules.mk`. It's content should be a list of XML input file names *without* the `.xml` file name suffix. For the document you are reading, the relevant line of the makefile is:

```
XMLSOURCE=build
```

The documentation support makefile fragments are only included if the `XMLSOURCE` variable is defined. By default, both HTML and PDF output are generated for all XML input documents.

The XML translation mechanism also knows how to generate `.eps` and `.gif` (soon switching to PNG) files from XFIG input. Figure files should be identified by the `FIGSOURCE` makefile variable. As with `XMLSOURCE`, this should be defined before `makerules.mk` is included:

```
FIGSOURCE=fig1.fig fig2.fig fig3.fig
```

We need better figure management system than XFIG, but the current system is better than nothing.

Because generating `.gif` and `.eps` files from the `.fig` sources is so fast, and because the output all goes in the same directory in any case, the current xml makefile fragments don't really attempt to associate individual figures to individual documents within a single XML document directory.

5.2 Building Object Code

In contrast to documentation, which is driven from the list of source files, object code compilation is driven from the list of target files. An example is provided below.

Several of the makefile variables shown deserve particular attention, because they control various aspects of the build process:

CROSS_BUILD States whether the build in this directory is done for the host (`CROSS_BUILD=no`) or for the target environment (`CROSS_BUILD=yes`). This variable needs to be declared before `makerules.mk` is included. It determines which set of compilers and build rules are used to build source files in the current directory.

OBJECTS The complete list of object files (`.o` files) that should be built in this directory. The **OBJECTS** variable is also used to determine the automatic dependency management files that need to be generated.

OPTIM The optimization level that should be applied. This defaults to `-O2`, but can be overridden locally.

IMGMAP Each service that is built into the system bootstrap image has an associated image map fragment (a `.imgmap` file). A corresponding header file is created for each image map file. The list of image map files, if any, should be given using the **IMGMAP** variable.

And here is an annotated example:

```
default: package

COYOTOS_SRC=../../..
# Use the CROSS_BUILD variable to say that this is a cross build.
# If absent, the build is assumed to be a host build.
CROSS_BUILD=yes

# Convention: binaries (or libraries) built in the current
# directory are named by the TARGETS variable.
TARGETS=$(BUILDDIR)/sample

# Convention: bootstrap image map files must be named by the
# IMGMAP variable. If x.imgmap is listed in the IMGMAP
# variable, then $(BUILDDIR)/x-constituents.h will be automatically generated
# by the automatic make rules.
IMGMAP=sample.imgmap

# Convention: all object files produced in this directory should
# be listed in the OBJECTS variable, and should
# be prefixed by $(BUILDDIR)/. If there are multiple targets, each
# should have its own specialized X_OBJECTS, Y_OBJECTS variables,
# which should be accumulated into OBJECTS
OBJECTS=$(BUILDDIR)/sample.o

# Default optimization level can be overridden by setting the value
# of OPTIM.
OPTIM=-O
```

```

# Include path can be modified by altering value of INC
INC=-I$(EROS_ROOT)/include
# Additional defines can be provided by altering value of DEFS
DEFS=-DSOMEDEF

include $(COYOTOS_SRC)/build/make/makerules.mk

# Convention: local 'all' target causes $(TARGETS) to be built.
# That is, 'all' is the local build target. The 'all' target
# should not recurse | it is assumed that the install
# target has already done that.
all: $(TARGETS)

# Link rule for the sample binary (this will need to change for
# Coyotos)
$(BUILDDIR)/sample: $(OBJECTS) $(DOMCRT0) $(DOMLIB)
$(DOMLINK) $(DOMLINKOPT) $(DOMCRT0) $(OBJECTS) -lsmall $(DOMLIB) -o $@

# Convention: 'install' target depends on 'all', installs
# targets into destination directory relative to COYOTOS_ROOT
install: all
$(INSTALL) -d $(COYOTOS_ROOT)/domain
$(INSTALL) -m 755 $(TARGETS) $(COYOTOS_ROOT)/domain
$(INSTALL) -m 644 $(IMGMAP) $(COYOTOS_ROOT)/domain

# Convention: last line of makefile should be the following, which
# ensures that existing automatic dependency files are reloaded
# after each execution.
-include $(BUILDDIR)/.*.m

```

Normally, the above mechanisms are sufficient. The `makerules.mk` file includes `makevars.mk`, which internally constructs values for the usual make variables `GCCFLAGS`, `GPLUSFLAGS` and so forth. Along the way, the value of `CROSS_BUILD` is used to decide how to set the values of variables `GCC`, `GPLUS`, `LD`, and friends. A more complete list of all variables associated with the build mechanism is certainly needed. For a better sense of what is going on, see `makevars.mk` and `build-rules.mk`. The variable names definitely need to be regularized and cleaned up.

The compilation mechanism is tuned to rebuild the associated automatic dependency files whenever an object file is rebuilt.