Creating Cross-Compile Friendly Software

Sam Robb
TimeSys
sam.robb@timesys.com

Abstract

Typical OSS packages make assumptions about their build environment that are not necessarily true when attempting to cross compile the software. There are two significant contributors to cross compile problems: platform specific code, and build/host confusion. Several examples of problems existing in current OSS packages are presented for each of these root causes, along with explanations of how they can be identified, how they can have been avoided, and how they can be resolved.

1 Why Cross Compile?

Cross compiling is the process of building software on a particular platform (architecture and operating system), with the intent of producing executables that will run on an entirely different platform. Generally, the platform the software is built on is referred to as the "build" system, while the platform the executables are run on is referred to as the "host" system.[1]

The process of cross compiling software is somewhat related to, but distinct from, the process of porting software to run on a different platform. The critical distinction is in the difference between the build and host system characteristics. Often times, software that can be built natively on different platforms will exhibit problems when cross compiling. These problems arise because the software fails to distinguish between the build system and the host system during one or more of the four distinct stages in the process of cross compiling software: configuration, compilation, installation, and verification.

Cross compiling is an absolute necessity for a very small number of software packages. In the OSS world, there are several software packages that are specifically designed with cross compiling in mind (binutils, gcc, busybox, the Linux kernel itself, etc.) These packages are often used to bootstrap a new system, providing a high-quality, low-cost way of obtaining a minimal working system with a small amount of effort. Once a minimal OS and related utilities are present on a system, a developer can then build additional software for the system as required.

As Linux becomes more prevalent in the embedded market space, there is an increased desire among embedded systems developers for more cross compile friendly software packages. While modern embedded systems are often resource rich in terms of processing power, I/O capabilities, memory, and disk space when compared to embedded systems of only a few years ago, compiling software natively on such a system still poses problems for an embedded developer. In extreme cases, compiling a moderately complex software package on an em-

[1] Unfortunately, not everyone chooses the same terminology. For example, the Scratchbox documentation (http://www.scratchbox.org/) uses the terms "host" and "target" where this paper uses "build" and "host" to refer to the same concepts.

bedded system natively may take hours instead of minutes.

Embedded developers therefore prefer cross compiling. Most significantly, it gives the embedded developer the advantage of working in a more comfortable, resource-rich environment—typically on a high-end workstation or desktop system—where they can take advantage of superior hardware to reduce their compile/link/debug cycles. Also importantly, cross compiling makes it easier to set up a system by which an entire system can easily be built from scratch in a reproducible manner.

2 Terminology and Assumptions

Cross compiling is a specialized subset of the software development world, and as such, employs its own terminology in an attempt unambiguously identify certain concepts. The following terms are definitions based on those provided by the GNU autoconf documentation [2], and used commonly in OSS projects such as binutils, gcc, etc.

platform - an architecture and OS combination

build system - the platform that a software package will be *configured* and *compiled* on

host system - the platform that a software package will *run* on

target system - the platform that the software package will *produce* output for

toolchain - the collection of tools (compiler, linker, etc.) along with the headers, libraries, etc. needed to build software for a platform

cross compiler - a **toolchain** that runs on a **host system**, but produces output for a **target system**

Typically, the target system is really only of interest to those working on compilers and related tools, where that extra degree of precision is needed in order to specify the final binary format those tools are intended to produce. In the OSS world, aside from binutils, gcc, and similar software packages, one can usually ignore the additional possibilities and complications introduced by variations in the target system.

The remainder of this paper will assume the existence of a cross compiler[3] that runs on an unspecified build system, and is capable of producing executables that will run on a different unspecified host system. The paper ignores the process of porting software to run on a new platform, in order to concentrate solely on issues that arise from the process of cross compiling the software.

3 Configuration Issues

All but the most simple software packages generally require some means of configuration. This is a process by which the software determines how it should be built—which libraries it should reference, which headers it may include, any particular quirks or workarounds in system calls it needs to deal with, etc.

Configuration is an area ripe for introducing cross compile problems. It provides software packages with the unique opportunity to completely confuse a build by assuming that the build system and the host system are one and the same. All cross compile configuration

[2] Available at `http://www.gnu.org/manual/`

[3] Those interested in building their own cross compiler may wish to consult the 'Resources' section at the end of this paper.

problems are some reflection of this confusion between the identity of the build and host systems.

3.1 Avoid using the wrong tools

This particular problem is caused by misidentifying which tools are to be used as part of the build process. Some software packages expect to be able to build and execute utility programs as part of their build process; a good example of this is the Linux kernel configuration utility. While the final output of the software package will need to run on the host system, these utility programs will need to be run on the build system.

Figure 1 shows an example of this problem. In this case, `CC_FOR_BUILD` is set to the same value as `CC`, which would be appropriate if it wasn't for the fact that earlier in the configuration process, `CC` was explicitly set to reference the cross compiler being used for the build.

```
 # compilers to use to create programs
 # which must be run in the build environment.
-CC_FOR_BUILD = $(CC)
-CXX_FOR_BUILD = $(CXX)
+CC_FOR_BUILD = gcc
+CXX_FOR_BUILD = g++

 SUBDIRS = "this is set via configure, \
           don't edit this"
 OTHERS =
```

Figure 1: Using the wrong tools

In this particular instance, there are several solutions. The most correct, and most expensive, is to update the makefile templates to use the proper variables (`CC_FOR_BUILD` and `CC`) in their proper context. Another possible solution is to override the definition of `CC_FOR_BUILD` and `CC` prior to invoking the makefile. The solution presented in Figure 1 is a simple, straightforward, get-it-working approach where `CC_FOR_BUILD` is simply set to an appropriate value for the majority of build systems.

3.2 Be cautious when executing code on the build system

As part of the configuration process, many software packages—particularly those built on top of `autoconf`—will try to compile, link, or even execute code on the host system.

For `autoconf` based projects, most of the standard `autoconf` macros (`AC_CHECK_LIB`, `AC_CHECK_HEADER`, etc.) do a good job of dealing with cross compile issues. In some instances, though, these standard macros fail when trying to test for the presence of an uncommon header file or library. Developers typically deal with these case by writing custom `autoconf` macros.

If the developer is not cautious, s/he may produce a custom macro that ends up performing a more extensive check than what is really needed. Often times, a developer will create a custom macro that makes use of the `autoconf AC_TRY_RUN` macro. This macro attempts to compile, link, and execute an arbitrary code fragment. The problem here is that the conditions being tested for may not actually require that the resulting binary be executed.

When cross compiling a package that uses custom macros, this leads to a situation where test code will compile and link properly (thanks to the cross compiler), but will then fail to run, or will run and produce incorrect output. In either case, it is highly unlikely that the configure script will reach the proper conclusion about whether or not the header file or library is actually available.

A simple solution to this problem is to check and see if the output from the test program is ever actually used. If not, then the call to `AC_TRY_RUN` in the test macro can be re-

placed with a call to AC_TRY_COMPILE or AC_TRY_LINK, as shown in Figure 2. These two macros implement checks for the ability to compile and link the provided code fragment, respectively.

```
      SKEY_MSG="yes"

      AC_MSG_CHECKING([for s/key support])
-     AC_TRY_RUN(
+     AC_TRY_LINK(
              [
 #include <stdio.h>
 #include <skey.h>
```

Figure 2: Avoiding execution when linking will suffice

3.3 Allow the user to override a 'detected' configuration value

In some cases, use of AC_TRY_RUN is absolutely essential; the automatic configuration process may need to be able to compile, link, and execute code in order to determine the characteristics of the host system. This is a definite stumbling block when trying to configure a software package for cross compiling.

A good configuration script allows the user to explicitly identify or override what would otherwise be an automatically detected value. For autoconf based projects, this typically means adding AC_ARG_ENABLE macros to your configure.in file that allow the user to explicitly set the value of questionable autoconf variables.

In the case of existing software packages, there may not be an explicit method for setting a questionable variable. In this case, it may be possible to set the appropriate variable by hand before configuring the software package, in order to force the desired outcome. This may still fail under some circumstances; for example, some configuration scripts do not bother to check to see if the a configuration variable has been set before attempting to automatically deduce its value.

In those cases, the configuration script may be modified[4] to guard the detection code by checking to see if the variable has already been assigned a value. If a value has already been assigned, the configuration script can use the specified value, and skip executing the detection code. In other cases, it may be more appropriate to fix the detection code itself so that it sets the variable to the proper value.

4 Compilation Issues

For the majority of portable software packages, attempting to cross compile will generally not uncover any issues with the code itself.[5] Even though individual source files may compile when pushed through the cross compiler, though, the overall way in which the software is built can still exhibit problems.

4.1 Avoid hard-coded tool names

Figure 4 shows a makefile fragment that originally made an explicit call to ar. In a package that is otherwise cross compile friendly, this is a particularly annoying occurrence. Depending on the specifics of the cross compiler, the call to ar may succeed, but produce an unusable static library.

Correcting this kind of problem is straightforward—replace the hard-coded tool name with a reference to a make variable

[4] For autoconf based software packages, keep in mind that the configure script is generated by processing configure.in. Editing the configure script direclty can be helpful for testing fixes, but changes will have to be made to configure.in as well to ensure they persist if the configure script is regenerated.

[5] Provided, of course, that the software has already been ported to the host platform.

that names the appropriate tool for the system the binary is intended to run on.

4.2 Avoid decorated tool names

Occaisionally, project makefiles will avoid hardcoded tool names by defining a variable, but then attempt to eliminate the an "unneeded" variable by combining a tool reference with the default flags that should be passed along to the tool, as shown in Figure 3.

While the intent was noble, this type of definition makes it difficult for a user to supply a different definition for a tool. Instead of simply setting the value of of the tool when invoking the makefile (ex, `make AR=ppc7xx-linux-ar`), a user now has to know to define AR in a way that includes the default arguments (ex, `make AR='ppc7xx-linux-ar cr'`).

Again, correcting this type of problem is straightforward—split the definition of the tool reference into a reference to the simple tool name and a variable that indicates the default flags that should be passed to the tool.

```
-AR = @AR@ cq
+AR = @AR@
+ARFLAGS = cq

 all: $(OBJS)
        -rm -f libsupport.a
-       $(AR) libsupport.a $(OBJS)
+       $(AR) $(ARFLAGS) libsupport.a $(OBJS)
        @RANLIB@ libsupport.a
```

Figure 3: Avoiding execution when linking will suffice

4.3 Avoid hard-coded paths

It is very easy for an otherwise cross compile friendly software package to mistakenly set up an absolute include path that looks reasonable. In many situations, the added include path may in fact be harmless, particularly if the build system and host system have roughly the same OS version, library versions, etc. However, even slight differences in structure definitions, enumerated constants, etc. between build system and host system headers can very easily result in either compilation errors, or in the cross compiler producing an unusable binary.

Figures 5 and 6 shows a simple and straightforward solution—remove the hard-coded include path. If the include path is required, then you will need to alter it so that it can be specified relative to the location of the include files appropriate for the host system.

4.4 Avoid assumptions about the build system

While this is nominally a porting issue, sometimes a software package will make what seems to be a reasonable assumption about the build system. In particular, software packages that are intended to run only on a particular class of operating systems (Linux, POSIX complaint systems, etc.) may assume that even if they are cross compiled, they will at least be cross compiled on a build system that has characteristics similar to the host system.

Figure 7 illustrates this problem. This makefile fragment assumes that the build system will have a case-sensitive file system, and that the file patterns `'*.os'` and `'*.oS'` will therefore refer to a distinct set of files—in this case, files for inclusion in a static library and files for inclusion in a shared library, respectively.

This particular assumption breaks down when compiling on a case-insensitive file system like VFAT, NTFS, or HPFS.[6] When encountering this type of problem, there is no easy workaround—the build logic for the software

[6] While these file systems are case-insensitive, they are case preserving, which sometimes helps mask potential case-sensitivity issues.

will need to be altered in order to adjust to the conditions of the unexpected build system.

In this case, the solution was to replace '`*.oS`' with '`*.on`', a file pattern that is distinct from '`*.os`' on either a case-insensitive or a case-sensitive file system.

5 Installation Issues

Software installation is sometimes seen as a simple problem. After all, how hard can it be to just copy files around and make sure they all end up in the right place? As with configuration and compilation, though, cross compiling software introduces additional complexities when installing software.

5.1 Avoid `install -s`

Figure 8 shows a makefile fragment that at first glance looks reasonable; as originally written, it attempted to install a binary using the detected version of the `install` program available on the build system.

The problem here is that the original `install` command specified the `-s` option, which instructs `install` to strip the binary after installing it. Because the command uses the build system's version of `install`, this means that the stripping will be accomplished using the build system's version of `strip`. Depending on the version of `strip` installed on the build system, this command may appear to succeed, yet result in a useless binary being installed.

The solution here is to avoid the use of `install -s`, and instead explicitly strip the binary after installation using the version of `strip` provided with the cross compile toolchain that built the binary.

5.2 Avoid hard-coded installation paths

When cross compiling software, it is often convenient to treat a directory on the build system as the logical root of the host system's file system.[7] This allows a developer to "install" the software into this logical root file system (RFS); often times, the RFS is made available to the host system via NFS.

Autoconf packages typically use variables to specify the prefix for installation paths, which makes installing them into an RFS a simple matter. As Figure 9 shows, non-`autoconf` makefiles may need to be modified to make the same sort of adjustments to installation paths.

Even if the software package already makes use of `prefix` or a similar variable, it may overload the meaning of that variable. This can happen in any type of software package, `autoconf` based or not. For example, a package may use the `prefix` variable to both control the installation path, and also generate `#define` statements that specify paths to configuration files or other important data. In this case, it may still be necessary to modify the makefile to introduce the idea of an installation prefix, as shown in Figure 10.

5.3 Create the required directory structure

Often times, software packages assume that they are being installed on an existing, full-featured system—which implies the existence of a certain directory structure. A cross compiled software package may be installed on the build system into a location that is lacking part or all of a normal directory structure. In this case, the install steps of the software package must be pessimistic, and assume that it will always be necessary to create whatever directory

[7] See the Scratchbox website (`http://www.scratchbox.org`) for more information on the hows and whys of build sandboxing.

structure it requires for the installation to succeed.

Figure 11 shows a patch for a makefile fragment that originally assumed the pre-existence of a particular directory structure. Appropriate calls to `mkdir -p` are enough to ensure that the existing directory structure is in place prior to the install.

6 Verification Issues

There are a number of OSS packages that very conveniently provide self-test capabilities. Along with the usual targets in their makefiles, they include targets that allow the user to build and run a test suite against the software after it is built, but before it is installed.

The main problem here is that these test targets generally run each individual test in the suite using a "compile, execute, analyze" cycle. Even if the compilation and result analysis steps succeed on the build system, test execution will most likely fail if the package has been cross compiled, since the tests were built with the host system in mind. If you are fortunate, these tests will simply fail; otherwise, you will not be able to gauge the accuracy of the tests, as they may be picking up information or artifacts from the build system.

A simple solution is to rewrite test targets to separate test compilation from test execution and result analysis. Providing a distinct install or packaging target for the test suite so that it can be easily moved over to a host system for execution is an added bonus.

Don't assume that you can execute self-tests as part of the normal build cycle (see Figure 12). If you do include a test target as part of your default target dependencies, at least make sure that it is only enabled or run if it knows that it can execute the tests on the build system.

7 Conclusions

By now, it should be apparent that while there are any number of subtle ways that cross compiling software can fail, they are for the most part simple problems with simple solutions.

Developers interested in supporting cross compiling of software packages they maintain can use these problems as a guideline of potential problem areas in their own projects. Detecting potential cross compile issues is often a simple matter of examining project source code and identifying the potential for confusing the meaning of build and host systems.

Finally—the best possible way to examine a software package to see if (or how well) it supports cross compiling is to actually try and cross compile it. While the truly adventurous may wish to try and build their own cross compiler, there are any number of locations on the web where an interested developer can obtain a pre-built toolchain for this purpose. Those working primarily on an x86 Linux host may wish to consider using one of the available pre-built cross compilers that can be found through the rpmfind (`http://www.rpmfind.net`) service. For those interested in building their own cross compiler, or in researching other cross compile issues, are a number of resources (see Table 8) on the net that deal specifically with cross compile issues. The emphasis of these resources is generally on embedded system development, though much of the information available is still applicable when discussing cross compiling in general.

8 Appendix—Code Examples

The following figures are referred to in the paper, and are collected here (instead of presented inline) for the sake of providing clarity in the text. Each figure represents a patch (or a partial patch) for a common OSS package that was used at TimeSys to work around cross compile problems. These selections were chosen to illustrate, in a compact fashion, both the problems described in the text and some possible solutions.

```
       decompress.o \
       bzlib.o

-all: libbz2.a bzip2 bzip2recover test
+all: libbz2.a bzip2 bzip2recover #test

 bzip2: libbz2.so bzip2.c
        $(CC) $(CFLAGS) -o bzip2 $\^
```

Figure 12: Avoid making tests part of the default build target

The CrossGCC Mailing List	`http://sources.redhat.com/ml/crossgcc/` A list for discussing embedded ('cross') programming using the GNU tools.
The CrossGCC FAQ	`http://www.sthoward.com/CrossGCC/`
crosstool	`http://www.kegel.com/crosstool/` A set of scripts to build gcc and glibc for most architectures supported by glibc.
Linux from Scratch	`http://www.linuxfromscratch.org/` A project that provides you with the steps necessary to build your own custom Linux system.
Scratchbox	`http://www.scratchbox.org/` A cross-compile toolkit for embedded Linux application development.
Embedded Gentoo	`http://www.gentoo.org/proj/en/base/embedded/index.xml` Gentoo project concerned with cross compiling and embedded systems.
The GNU configure and build system	`http://www.airs.com/ian/configure/` Document describing the GNU configure and build systems. A bit out of date (circa 1998), but still very useful.
GNU Autoconf, Automake, and Libtool	`http://sources.redhat.com/autobook/` Online version of the classic book covering GNU autotools.

Table 1: Selected internet resources on cross compiling

```
 libbz2.a: $(OBJS)
        rm -f libbz2.a
-       ar cq libbz2.a $(OBJS)
-       @if ( test -f /usr/bin/ranlib -o -f /bin/ranlib -o \
-               -f /usr/ccs/bin/ranlib ) ; then \
-               echo ranlib libbz2.a ; \
-               ranlib libbz2.a ; \
-       fi
+       $(AR) cq libbz2.a $(OBJS)
+       $(RANLIB) libbz2.a
+       #@if ( test -f /usr/bin/ranlib -o -f /bin/ranlib -o \
+       #       -f /usr/ccs/bin/ranlib ) ; then \
+       #       echo ranlib libbz2.a ; \
+       #       ranlib libbz2.a ; \
+       #fi

 libbz2.so: libbz2.so.$(somajor)
```

Figure 4: Avoiding hard-coded tool references

```
 export GCC_WARN   =  -Wall -W -Wstrict-prototypes -Wshadow $(ANAL_WARN)
-export INCDIRS    =  -I/usr/include/ncurses
-export CC         =  gcc
+#export INCDIRS    =  -I/usr/include/ncurses
+#export CC         =  gcc
 export OPT        =  -O2
 export CFLAGS     =  -D_GNU_SOURCE $(OPT) $(GCC_WARN) -I$(shell pwd) $(INCDIRS)
```

Figure 5: Avoiding hard-coded include paths

```
 INSTALL        = install -o $(BIN_OWNER) -g $(BIN_GROUP)

 # Additional libs for Gnu Libc
-ifneq ($(wildcard /usr/lib/libcrypt.a),)
 LCRYPT         = -lcrypt
-endif

 all:           $(PROGS)
```

Figure 6: Avoiding tests for hard-coded path names

```
 # Bounded pointer thunks are only built for *.ob
 elide-bp-thunks = $(addprefix $(bppfx),$(bp-thunks))

-elide-routines.oS += $(filter-out $(static-only-routines),\
+elide-routines.on += $(filter-out $(static-only-routines),\
    $(routines) $(aux) $(sysdep_routines)) \
       $(elide-bp-thunks)
 elide-routines.os += $(static-only-routines) $(elide-bp-thunks)
```

Figure 7: Avoiding assumptions about the build system

```
-       $(INSTALL) -m 0755 -s ssh $(DESTDIR)$(bindir)/ssh
+       $(INSTALL) -m 0755 ssh $(DESTDIR)$(bindir)/ssh
+       $(STRIP) $(DESTDIR)$(bindir)/ssh
```

Figure 8: Replacing install -s with an explicit call to strip

```
 NAME       =  proc

 # INSTALLATION OPTIONS
-TOPDIR     = /usr
+TOPDIR     = $(DESTDIR)/usr
 HDRDIR     = $(TOPDIR)/include/$(NAME)#        where to put .h files
 LIBDIR     = $(TOPDIR)/lib#       where to put library files
-SHLIBDIR   = /lib#                where to put shared library files
+SHLIBDIR   = $(DESTDIR)/lib#      where to put shared library files
 HDROWN     = $(OWNERGROUP) #      owner of header files
 LIBOWN     = $(OWNERGROUP) #      owner of library files
 INSTALL    = install
```

Figure 9: Avoiding hard-coded install paths

```
 # Where is include and dir located?
 prefix=/
+installdir=/

 .c.o:
        $(CC) $(CFLAGS) -c $<
@@ -47,28 +48,32 @@
        -if [ ! -d pic ]; then mkdir pic; fi

 install: lib install-dirs install-data
-       -if [ -f $(prefix)/lib/$(SHARED_LIB) ]; then \
-          mkdir -p $(prefix)/lib/backup; \
-          mv $(prefix)/lib/$(SHARED_LIB) \
-               $(prefix)/lib/backup/$(SHARED_LIB).$$$$; \
+       -if [ -f $(installdir)/$(prefix)/lib/$(SHARED_LIB) ]; then \
+          mkdir -p $(installdir)/$(prefix)/lib/backup; \
+          mv $(installdir)/$(prefix)/lib/$(SHARED_LIB) \
+               $(installdir)/$(prefix)/lib/backup/$(SHARED_LIB).$$$$; \
        fi
-       cp $(SHARED_LIB) $(prefix)/lib
-       chown $(OWNER) $(prefix)/lib/$(SHARED_LIB)
+       cp $(SHARED_LIB) $(installdir)/$(prefix)/lib
+       chown $(OWNER) $(installdir)/$(prefix)/lib/$(SHARED_LIB)
        if [ -x /sbin/ldconfig -o -x /etc/ldconfig ]; then \
          ldconfig; \
```

Figure 10: Working around the use of an overloaded prefix variable

```
 install-only:
   n=‘echo gdbserver | sed ’$(program_transform_name)’‘; \
   if [ x$$n = x ]; then n=gdbserver; else true; fi; \
+  mkdir -p $(bindir); \
+  mkdir -p $(man1dir); \
   $(INSTALL_PROGRAM) gdbserver $(bindir)/$$n; \
   $(INSTALL_DATA) $(srcdir)/gdbserver.1 $(man1dir)/$$n.1
```

Figure 11: Creating required directories at install time

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*