Bash by example, Part 3

Exploring the ebuild system

Daniel Robbins
President and CEO
Gentoo Technologies, Inc.

01 May 2000

In his final *Bash by example* article, Daniel Robbins takes a good look at the Gentoo Linux ebuild system, an excellent example of the power of bash. Step by step, he shows you how the ebuild system was implemented, and touches on many handy bash techniques and design strategies. By the end of the article, you'll have a good grasp of what's involved in producing a full-blown bash-based application, as well as a start at coding your own auto-build system.

Enter the ebuild system

I've really been looking forward to this third and final *Bash by example* article, because now that we've already covered bash programming fundamentals in Part 1 and Part 2, we can focus on more advanced topics, like bash application development and program design. For this article, I will give you a good dose of practical, real-world bash development experience by presenting a project that I've spent many hours coding and refining: The Gentoo Linux ebuild system.

I'm the chief architect of Gentoo Linux, a next-generation Linux OS currently in beta. One of my primary responsibilities is to make sure that all of the binary packages (similar to RPM packages) are created properly and work together. As you probably know, a standard Linux system is not composed of a single unified source tree (like BSD), but is actually made up of about 25+ core packages that work together. Some of the packages include:

Package	Description
linux	The actual kernel
util-linux	A collection of miscellaneous Linux-related programs
e2fsprogs	A collection of ext2 filesystem-related utilities
glibc	The GNU C library

Each package is in its own tarball and is maintained by separate independent developers, or teams of developers. To create a distribution, each package has to be separately downloaded, compiled, and packaged. Every time a package must be fixed, upgraded, or improved, the compilation and packaging steps must be repeated (and this gets old really fast). To help eliminate the repetitive steps involved in creating and updating packages, I created the ebuild system, written almost entirely in bash. To enhance your bash knowledge, I'll show you how I implemented the unpack and compile portions of the ebuild system, step by step. As I explain each step, I'll also discuss why certain design decisions were made. By the end of this article, not only will you have an excellent grasp of larger-scale bash programming projects, but you'll also have implemented a good portion of a complete auto-build system.

Why bash?

Bash is an essential component of the Gentoo Linux ebuild system. It was chosen as ebuild's primary language for a number of reasons. First, it has an uncomplicated and familiar syntax that is especially well suited for calling external programs. An auto-build system is "glue code" that automates the calling of external programs, and bash is very well suited to this type of application. Second, Bash's support for functions allowed the ebuild system to have modular, easy-to-understand code. Third, the ebuild system takes advantage of bash's support for environment variables, allowing package maintainers and developers to configure it easily, on-the-fly.

Build process review

Before we look at the ebuild system, let's review what's involved in getting a package compiled and installed. For our example, we will look at the "sed" package, a standard GNU text stream editing utility that is part of all Linux distributions. First, download the source tarball (sed-3.02.tar.gz) (see Resources). We will store this archive in /usr/src/distfiles, a directory we will refer to using the environment variable "$DISTDIR". "$DISTDIR" is the directory where all of our original source tarballs live; it's a big vault of source code.

Our next step is to create a temporary directory called "work", which houses the uncompressed sources. We'll refer to this directory later using the "$WORKDIR" environment variable. To do this, change to a directory where we have write permission and type the following:

Uncompressing sed into a temporary directory

```
$ mkdir work
$ cd work
$ tar xzf /usr/src/distfiles/sed-3.02.tar.gz
```

The tarball is then decompressed, creating a directory called sed-3.02 that contains all of the sources. We'll refer to the sed-3.02 directory later using the environment variable "$SRCDIR". To compile the program, type the following:

Uncompressing sed into a temporary directory

```
$ cd sed-3.02
$ ./configure --prefix=/usr
(autoconf generates appropriate makefiles, this can take a while)

$ make

(the package is compiled from sources, also takes a bit of time)
```

We're going to skip the "make install" step, since we are just covering the unpack and compile steps in this article. If we wanted to write a bash script to perform all these steps for us, it could look something like this:

Sample bash script to perform the unpack/compile process

```
#!/usr/bin/env bash

if [ -d work ]
then
# remove old work directory if it exists
        rm -rf work
fi
mkdir work
cd work
tar xzf /usr/src/distfiles/sed-3.02.tar.gz
cd sed-3.02
./configure --prefix=/usr
make
```

Generalizing the code

Although this autocompile script works, it's not very flexible. Basically, the bash script just contains the listing of all the commands that were typed at the command line. While this solution works, it would be nice to make a generic script that can be configured quickly to unpack and compile any package just by changing a few lines. That way, it's much less work for the package maintainer to add new packages to the distribution. Let's take a first stab at doing this by using lots of different environment variables, making our build script more generic:

A new, more general script

```
#!/usr/bin/env bash

# P is the package name

P=sed-3.02

# A is the archive name

A=${P}.tar.gz

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work
export SRCDIR=${WORKDIR}/${P}

if [ -z "$DISTDIR" ]
then
        # set DISTDIR to /usr/src/distfiles if not already set
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

if [ -d ${WORKDIR} ]
then
        # remove old work directory if it exists
        rm -rf ${WORKDIR}
fi

mkdir ${WORKDIR}
cd ${WORKDIR}
tar xzf ${DISTDIR}/${A}
cd ${SRCDIR}
```

```
./configure --prefix=/usr
make
```

We've added a lot of environment variables to the code, but it still does basically the same thing. However, now, to compile any standard GNU autoconf-based source tarball, we can simply copy this file to a new file (with an appropriate name to reflect the name of the new package it compiles), and then change the values of "$A" and "$P" to new values. All other environment variables automatically adjust to the correct settings, and the script works as expected. While this is handy, there's a further improvement that can be made to the code. This particular code is much longer than the original "transcript" script that we created. Since one of the goals for any programming project should be the reduction of complexity for the user, it would be nice to dramatically shrink the code, or at least organize it better. We can do this by performing a neat trick -- we'll split the code into two separate files. Save this file as "sed-3.02.ebuild":

sed-3.02.ebuild

```
#the sed ebuild file -- very simple!
P=sed-3.02
A=${P}.tar.gz
```

Our first file is trivial, and contains only those environment variables that must be configured on a per-package basis. Here's the second file, which contains the brains of the operation. Save this one as "ebuild" and make it executable:

The ebuild script

```
#!/usr/bin/env bash


if [ $# -ne 1 ]
then
        echo "one argument expected."
        exit 1
fi

if [ -e "$1" ]
then
        source $1
else
        echo "ebuild file $1 not found."
        exit 1
fi

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work
export SRCDIR=${WORKDIR}/${P}

if [ -z "$DISTDIR" ]
then
        # set DISTDIR to /usr/src/distfiles if not already set
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

if [ -d ${WORKDIR} ]
then
        # remove old work directory if it exists
        rm -rf ${WORKDIR}
fi
```

```
mkdir ${WORKDIR}
cd ${WORKDIR}
tar xzf ${DISTDIR}/${A}
cd ${SRCDIR}
./configure --prefix=/usr
make
```

Now that we've split our build system into two files, I bet you're wondering how it works. Basically, to compile sed, type:

```
$ ./ebuild sed-3.02.ebuild
```

When "ebuild" executes, it first tries to "source" variable "$1". What does this mean? From my previous article, recall that "$1" is the first command line argument -- in this case, "sed-3.02.ebuild". In bash, the "source" command reads in bash statements from a file, and executes them as if they appeared immediately in the file the "source" command is in. So, "source ${1}" causes the "ebuild" script to execute the commands in "sed-3.02.ebuild", which cause "$P" and "$A" to be defined. This design change is really handy, because if we want to compile another program instead of sed, we can simply create a new .ebuild file and pass it as an argument to our "ebuild" script. That way, the .ebuild files end up being really simple, while the complicated brains of the ebuild system get stored in one place -- our "ebuild" script. This way, we can upgrade or enhance the ebuild system simply by editing the "ebuild" script, keeping the implementation details outside of the ebuild files. Here's a sample ebuild file for gzip:

gzip-1.2.4a.ebuild

```
#another really simple ebuild script!
P=gzip-1.2.4a
A=${P}.tar.gz
```

Adding functionality

OK, we're making some progress. But, there is some additional functionality I'd like to add. I'd like the ebuild script to accept a second command-line argument, which will be "compile", "unpack", or "all". This second command-line argument tells the ebuild script which particular step of the build process to perform. That way, I can tell ebuild to unpack the archive, but not compile it (just in case I need to inspect the source archive before compilation begins). To do this, I'll add a case statement that will test variable "$2", and do different things based on its value. Here's what the code looks like now:

ebuild, revision 2

```
#!/usr/bin/env bash

if [ $# -ne 2 ]
then
        echo "Please specify two args - .ebuild file and unpack, compile or all"
        exit 1
fi


if [ -z "$DISTDIR" ]
then
        # set DISTDIR to /usr/src/distfiles if not already set
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

ebuild_unpack() {
        #make sure we're in the right directory
        cd ${ORIGDIR}

        if [ -d ${WORKDIR} ]
        then
                rm -rf ${WORKDIR}
        fi

        mkdir ${WORKDIR}
        cd ${WORKDIR}
        if [ ! -e ${DISTDIR}/${A} ]
        then
            echo "${DISTDIR}/${A} does not exist.  Please download first."
            exit 1
        fi
        tar xzf ${DISTDIR}/${A}
        echo "Unpacked ${DISTDIR}/${A}."
        #source is now correctly unpacked
}


ebuild_compile() {

        #make sure we're in the right directory
        cd ${SRCDIR}
        if [ ! -d "${SRCDIR}" ]
        then
                echo "${SRCDIR} does not exist -- please unpack first."
                exit 1
        fi
        ./configure --prefix=/usr
        make
}

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work

if [ -e "$1" ]
then
        source $1
else
        echo "Ebuild file $1 not found."
        exit 1
fi

export SRCDIR=${WORKDIR}/${P}

case "${2}" in
        unpack)
                ebuild_unpack
                ;;
        compile)
                ebuild_compile
                ;;
        all)
                ebuild_unpack
                ebuild_compile
                ;;
        *)
                echo "Please specify unpack, compile or all as the second arg"
                exit 1
                ;;
```

```
esac
```

We've made a lot of changes, so let's review them. First, we placed the compile and unpack steps in their own functions, and called ebuild_compile() and ebuild_unpack(), respectively. This is a good move, since the code is getting more complicated, and the new functions provide some modularity, which helps to keep things organized. On the first line in each function, I explicitly "cd" into the directory I want to be in because, as our code is becoming more modular rather than linear, it's more likely that we might slip up and execute a function in the wrong current working directory. The "cd" commands explicitly put us in the right place, and prevent us from making a mistake later -- an important step -- especially if you will be deleting files inside the functions.

Also, I added a useful check to the beginning of the ebuild_compile() function. Now, it checks to make sure the "$SRCDIR" exists, and, if not, it prints an error message telling the user to unpack the archive first, and then exits. If you like, you can change this behavior so that if "$SRCDIR" doesn't exist, our ebuild script will unpack the source archive automatically. You can do this by replacing ebuild_compile() with the following code:

A new spin on ebuild_compile()

```
ebuild_compile() {
        #make sure we're in the right directory
        if [ ! -d "${SRCDIR}" ]
        then
                ebuild_unpack
        fi
        cd ${SRCDIR}
        ./configure --prefix=/usr
        make
}
```

One of the most obvious changes in our second version of the ebuild script is the new case statement at the end of the code. This case statement simply checks the second command-line argument, and performs the correct action, depending on its value. If we now type:

```
$ ebuild sed-3.02.ebuild
```

we'll actually get an error message. ebuild now wants to be told what to do, as follows:

```
$ ebuild sed-3.02.ebuild unpack
```

or

```
$ ebuild sed-3.02.ebuild compile
```

or

```
$ ebuild sed-3.02.ebuild all
```

If you provide a second command-line argument, other than those listed above, you get an error message (the * clause), and the program exits.

Modularizing the code

Now that the code is quite advanced and functional, you may be tempted to create several more ebuild scripts to unpack and compile your favorite programs. If you do, sooner or later you'll come across some sources that do not use autoconf ("./configure") or possibly others that have non-standard compilation processes. We need to make some more changes to the ebuild system to accommodate these programs. But before we do, it is a good idea to think a bit about how to accomplish this.

One of the great things about hard-coding "./configure --prefix=/usr; make" into our compile stage is that, most of the time, it works. But, we must also have the ebuild system accommodate sources that do not use autoconf or normal Makefiles. To solve this problem, I propose that our ebuild script should, by default, do the following:

1. If there is a configure script in "${SRCDIR}", execute it as follows:
   `./configure --prefix=/usr`
   Otherwise, skip this step.
2. Execute the following command: `make`

Since ebuild only runs configure if it actually exists, we can now automatically accommodate those programs that don't use autoconf and have standard makefiles. But what if a simple "make" doesn't do the trick for some sources? We need a way to override our reasonable defaults with some specific code to handle these situations. To do this, we'll transform our ebuild_compile() function into two functions. The first function, which can be looked at as a "parent" function, will still be called ebuild_compile(). However, we'll have a new function, called user_compile(), which contains only our reasonable default actions:

ebuild_compile() split into two functions

```
user_compile() {
        #we're already in ${SRCDIR}
        if [ -e configure ]
        then
                #run configure script if it exists
                ./configure --prefix=/usr
        fi
        #run make
        make
}

ebuild_compile() {
        if [ ! -d "${SRCDIR}" ]
        then
                echo "${SRCDIR} does not exist -- please unpack first."
                exit 1
        fi
        #make sure we're in the right directory
        cd ${SRCDIR}
        user_compile
}
```

It may not seem obvious why I'm doing this right now, but bear with me. While the code works almost identically to our previous version of ebuild, we can now do something that we couldn't do before -- we can override user_compile() in sed-3.02.ebuild. So, if the default user_compile() function doesn't meet our needs, we can define a new one in our .ebuild file that contains the commands required to compile the package. For example, here's an ebuild file for e2fsprogs-1.18, which requires a slightly different "./configure" line:

e2fsprogs-1.18.ebuild

```
#this ebuild file overrides the default user_compile()
P=e2fsprogs-1.18
A=${P}.tar.gz

user_compile() {
       ./configure --enable-elf-shlibs
       make
}
```

Now, e2fsprogs will be compiled exactly the way we want it to be. But, for most packages, we can omit any custom user_compile() function in the .ebuild file, and the default user_compile() function is used instead.

How exactly does the ebuild script know which user_compile() function to use? This is actually quite simple. In the ebuild script, the default user_compile() function is defined before the e2fsprogs-1.18.ebuild file is sourced. If there is a user_compile() in e2fsprogs-1.18.ebuild, it overwrites the default version defined previously. If not, the default user_compile() function is used.

This is great stuff; we've added a lot of flexibility without requiring any complex code if it's not needed. We won't cover it here, but you could also make similar modifications to ebuild_unpack() so that users can override the default unpacking process. This could come in handy if any patching has to be done, or if the files are contained in multiple archives. It is also a good idea to modify our unpacking code so that it recognizes bzip2-compressed tarballs by default.

Configuration files

We've covered a lot of sneaky bash techniques so far, and now it's time to cover one more. Often, it's handy for a program to have a global configuration file that resides in /etc. Fortunately, this is easy to do using bash. Simply create the following file and save it as /etc/ebuild.conf:

/ect/ebuild.conf

```
# /etc/ebuild.conf: set system-wide ebuild options in this file

# MAKEOPTS are options passed to make
MAKEOPTS="-j2"
```

What is a parallel make?

To speed compilation on multiprocessor systems, make supports compiling a program in parallel. This means that instead of compiling just one source file at a time, make compiles a user-specified number of source files simultaneously (so those extra processors in a multiprocessor system are used). Parallel makes are enabled by passing the -j # option to make, as follows:

```
make -j4 MAKE="make -j4"
```

This code instructs make to compile four programs simultaneously. The `MAKE="make -j4"` argument tells make to pass the -j4 option to any child make processes it launches.

In this example, I've included just one configuration option, but you could include many more. One of the beautiful things about bash is that this file can be parsed by simply sourcing it. This is a design trick that works with most interpreted languages. After /etc/ebuild.conf is sourced, "$MAKEOPTS" is defined inside our ebuild script. We'll use it to allow the user to pass options to make. Normally, this option would be used to allow the user to tell ebuild to do a parallel make.

Here's the final version of our ebuild program:

ebuild, the final version

```
#!/usr/bin/env bash

if [ $# -ne 2 ]
then
        echo "Please specify ebuild file and unpack, compile or all"
        exit 1
fi

source /etc/ebuild.conf

if [ -z "$DISTDIR" ]
then
        # set DISTDIR to /usr/src/distfiles if not already set
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

ebuild_unpack() {
        #make sure we're in the right directory
        cd ${ORIGDIR}

        if [ -d ${WORKDIR} ]
        then
                rm -rf ${WORKDIR}
        fi

        mkdir ${WORKDIR}
        cd ${WORKDIR}
        if [ ! -e ${DISTDIR}/${A} ]
        then
                echo "${DISTDIR}/${A} does not exist.  Please download first."
                exit 1
        fi
        tar xzf ${DISTDIR}/${A}
        echo "Unpacked ${DISTDIR}/${A}."
        #source is now correctly unpacked
}

user_compile() {
        #we're already in ${SRCDIR}
        if [ -e configure ]
        then
                #run configure script if it exists
                ./configure --prefix=/usr
        fi
        #run make
        make $MAKEOPTS MAKE="make $MAKEOPTS"
}
```

```
ebuild_compile() {
        if [ ! -d "${SRCDIR}" ]
        then
                echo "${SRCDIR} does not exist -- please unpack first."
                exit 1
        fi
        #make sure we're in the right directory
        cd ${SRCDIR}
        user_compile
}

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work

if [ -e "$1" ]
then
        source $1
else
        echo "Ebuild file $1 not found."
        exit 1
fi

export SRCDIR=${WORKDIR}/${P}

case "${2}" in
        unpack)
                ebuild_unpack
                ;;
        compile)
                ebuild_compile
                ;;
        all)
                ebuild_unpack
                ebuild_compile
                ;;
        *)
                echo "Please specify unpack, compile or all as the second arg"
                exit 1
                ;;
esac
```

Notice /etc/ebuild.conf is sourced near the beginning of the file. Also, notice that we use "$MAKEOPTS" in our default user_compile() function. You may be wondering how this will work -- after all, we refer to "$MAKEOPTS" before we source /etc/ebuild.conf, which actually defines "$MAKEOPTS" in the first place. Fortunately for us, this is OK because variable expansion only happens when user_compile() is executed. By the time user_compile() is executed, /etc/ebuild.conf has already been sourced, and "$MAKEOPTS" is set to the correct value.

Wrapping it up

We've covered a lot of bash programming techniques in this article, but we've only touched the surface of the power of bash. For example, the production Gentoo Linux ebuild system not only automatically unpacks and compiles each package, but it can also:

- Automatically download the sources if they are not found in "$DISTDIR"
- Verify that the sources are not corrupted by using MD5 message digests
- If requested, install the compiled application into the live filesystem, recording all installed files so that the package can be easily uninstalled at a later date

- If requested, package the compiled application in a tarball (compressed the way you like it) so that it can be installed later, on another computer, or during the CD-based installation process (if you are building a distribution CD)

In addition, the production ebuild system has several other global configuration options, allowing the user to specify options such as what optimization flags to use during compilation, and whether optional support for packages like GNOME and slang should be enabled by default in those packages that support it.

It's clear that bash can accomplish much more than what I've touched on in this series of articles. I hope you've learned a lot about this incredible tool, and are excited about using bash to speed up and enhance your development projects.

Resources

- Download the source tarball (sed-3.02.tar.gz) from ftp://ftp.gnu.org/pub/gnu/sed.
- Read "Bash by example: Part 1" on *developerWorks*.
- Read "Bash by example: Part 2" on *developerWorks*.
- Visit the home page of the Gentoo Project.
- Visit GNU's bash home page.

About the author

Daniel Robbins

Residing in Albuquerque, New Mexico, Daniel Robbins is the Chief Architect of the Gentoo Project, CEO of Gentoo Technologies, Inc., the mentor for the Linux Advanced Multimedia Project (LAMP), and a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, who is expecting a child this spring. You can contact Daniel at drobbins@gentoo.org.