From Source Code to the Finished Product: Understanding configure Error Messages

Perfect Configuration

There is no better place to view error messages than when compiling a program. Configure scripts are especially critical – time to find a way out of the error-prone labyrinth. At first it might seem full of bugs, but the application you want to build may be easier to fix that you would otherwise expect.

BY ANDREA MÜLLER

www.photocase.de

This is something that every Linux user will experience sooner or later. You are unable to find a binary package for that much sought after program for your distribution. To the Linux geeks the answer is obvious: "Compile it yourself!". Unfortunately, if the compiler is still unexplored territory for you, your efforts may lead to a mass of cryptic error messages, rather than an executable application. Some people call it a day at this stage, concluding that you need programming skills to successfully compile a program.

Fortunately, that is not true. The compilation process for most applications is automated for the most part. Read on, if you want to know more about this process, and what exactly happens during the three steps: *./configure*, *make*, and *make install*.

In this article, we will be looking at the error messages you may encounter when running *./configure*. To compile software correctly, it is important to know how to resolve these errors.

Basic Equipment

After downloading and unpacking the source code for an application, you need to work in the directory that this step creates. Take a look at the *README* and *INSTALL* files before you do anything else. These files tell you if it is possible to use the normal three card trick to build the program, or if you need to edit a file first. Also, many programmers use these files to inform users of additional software dependencies.

If this is the first time you have attempted to run *./configure* on your system, the **shell script** might quit with the following errors shortly afterwards:

```
checking for gcc... no
```

Listing 1: Missing developer package errors

```
01 [andi@diabolos squaroid-0.60.3]./configure
02 [...]
03 checking for GTK - version >= 1.2.0... yes
04 checking for imlib-config... no
05 checking for gdk_imlib... not found
06 configure: error: Cannot find gdk_imlib: I NEED IT!
```

GLOSSARY

./: *As the directory with the unpacked source code does not belong to the shell path, you cannot simply call the "configure" script below that directory by name. Instead, you need to supply the directory path, which is "./" for the current working directory.*

Shell script: *Text file with commands to be processed sequentially by the command interpreter (the shell – this tends to be Bash on most Linux distributions).*

locate: *This command line tool searches a database for the location of a file on disk. The database is created by the "updatedb" command, for which most distributions set up a daily cron job. On Suse Linux, the "updatedb"/"locate" combination is part of the "findutils-locate" package, and not installed by default.*

```
checking for cc... no
checking for cc... no
checking for cl... no
configure: error: no
acceptable ➤
C compiler found ➤
in $PATH
```

The script checks whether the files required to compile the software really exist. The lines starting with *checking* tell you what it checks. The results are shown at the end of each line, *no* in this case. In other words, the script quits and outputs an error message if it fails to find what it is looking for. In this case, the machine simply does not have a suitable compiler in its *$PATH*.

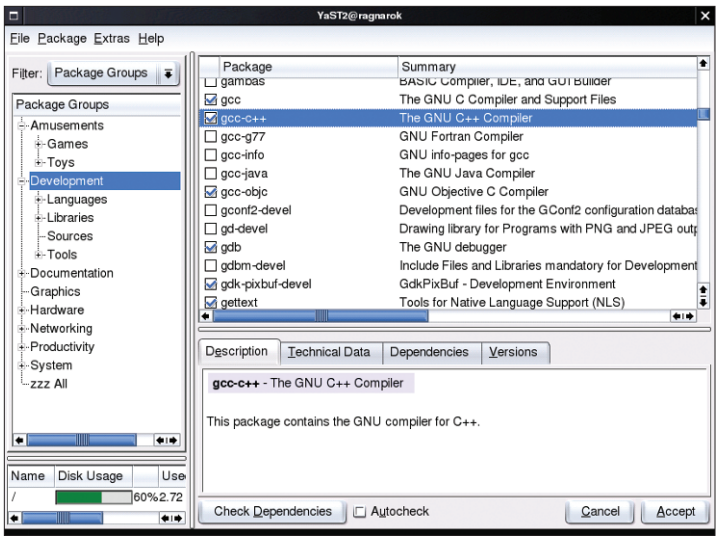

Figure 1: The "Development" package group category in Suse's YaST tool contains the packages needed for compiling.

The *PATH* variable (variables are indicated by the dollar sign, an operator that queries the value of the variable) contains the search path for executables. Every distribution has a different set of directories in its path. The *echo $PATH* command outputs a list of the directories in the path.

In this case, the shell was unable to find a C compiler, an application that converts software written in the C programming language into an executable. Linux typically uses the *gcc*, although most distributions no longer install this indispensable tool by default.

Table 1 has a list of packages that you need to get your system ready for compiling. These include tools such as the compiler and the *make* program, and developer packages with the files for compiling graphical applications.

Don't let the package descriptions for the devel packages confuse you. They imply that you only need these files if you want to write your own software. This is not true, as they are also necessary to compile some source code packages.

Use your distribution's package manager to install the developer packages (see Figure 1). If you have the Personal edition of Suse Linux, you will need to download most of these from the Suse FTP server first. The Personal edition has very little in the line of developer packages by default.

This basic set of tools will remove the cause of most error messages. If error messages continue to occur, you may need to do some detective work.

Seek, and Ye Shall Find

Listing 1 shows the most common type of *configure* error message.

In our example, the configuration script complains about a missing *gdk_imlib* (your target application may be missing *krb5.h*, or *lidSDL.so*). The line that says *checking for imlib-config... no* tells you what the script is looking for. Here, *configure* is looking for a script file called *imlib-config*. As it fails to find the script, – *no* –, it decides that *gdk_imlib* does not exist on the developer machine (*checking for gdk_imlib... not found*).

You can type ***locate*** *imlib-config* to find out whether the *imlib-config* file is missing on your machine, or if *configure* looked in the wrong places. Alternatively, use *find*, although this may take a lot longer, as it does not perform a database search.

If the search fails to come up with the goods, you can assume that the file really does not exist. In this case, you will need to install it. How do you go about locating the file in the thousands of packages that a distribution comprises? The tools provided by the Mandrake and Suse Linux distributions make this easy. If you have Mandrake, type *urpmf imlib-config*. This tells the package manager to consult the database and show you the files and packages containing *imlib-config* (see Figure 2). *urpmf gdk_imlib* tells us that the devel-

Table 1: Requirements for a developer system

Package	Contains
gcc	Compiler for software written in C programming language.
gcc-c++	Compiler for software written in C++ programming language.
libstdc++-devel	Developer files for compiling C++ programs.
make	Program for automated compilation of source code based on a "ruleset" provided by the *Makefile*.
binutils	Programs for manipulating binary files, including the archiving tool *ar*, and *strip*, which removes debugging information from programs and libraries.
glibc-devel	C library developer files.
gettext and gettext-devel	Programs and files for compiling multi-language software. Some distros do not have a *gettext-devel* package, as *gettext* includes the required files.
XFree86-devel (libxfree86-devel for some distros)	Developer files for the graphical system.
libpng-devel	Developer files from a library that displays PNG formatted images. Almost every graphical application will need this library.
libjpeg-devel	Developer files from the JPEG library.
zlib-devel	Developer files from the *zlib* compression library.
gtk-devel	*gtk* developer files. Required to compile *gtk1.x* applications, such as Sylpheed, a mail program.
gtk2-devel	*gtk2* developer files. Required to compile all *gtk2* programs, such as Gimp 2.0.
kdelibs-devel (kdelibs3-devel for KDE 3.x)	KDE library developer files; minimum requirement for compiling KDE programs.
qt3-devel (libqt3-devel for some distros)	Qt library developer, which provide GTK elements for an application GUI, just like GTK. Required to compile KDE and Qt applications.

Some distributions (e.g. Debian) use the *dev* suffix, rather than *devel*, to identify developer packages.

oper files are in the *libimlib1-devel* package.

Suse Linux users can run *pin* instead. When you first run the tool, it prompts you to insert the installation medium, from which it copies the package overview to your disk. You need administrative privileges to do this. After completing this step, you can enter *pin imlib-config* to discover that the file is in a package of the same name. *pin gdk_imlib* comes up with *imlib-devel*. On Suse Linux, you need to install both packages to stop *configure* outputting an error message and quitting.

Things are more difficult if you have the Personal version of Suse or Red Hat. Red Hat does not have a tool that allows you to search the file list for non-installed packages. Searching for developer packages on Suse personal tends to draw a blank, as the required packages are not normally on the CDs.

In both these cases, the search engine for RPM and Debian packages at [1] can be a useful alternative. The web form also checks for individual files and displays packages containing those files (see Figure 3). Debian users can also check out *http://packages.debian.org/*.

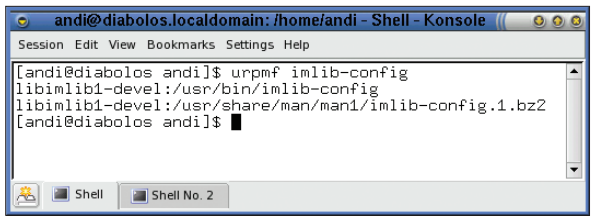

Figure 2: Mandrake's "urpmf" tool searches a database with a list of RPM packages belonging to the distro.

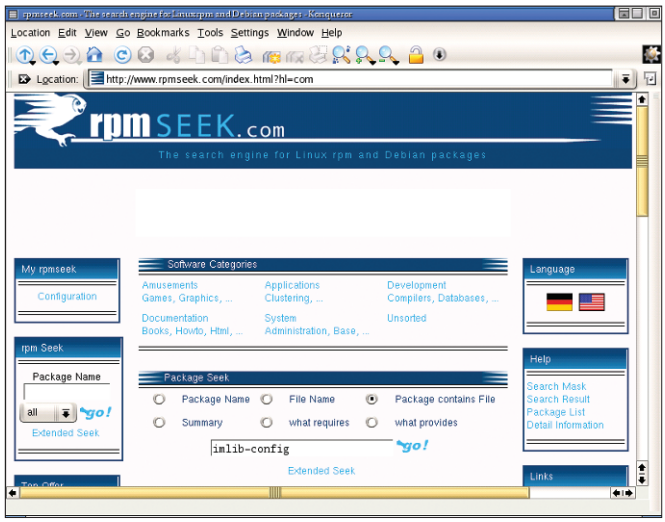

Figure 3: Rpmseek.com not only searches for packages, but will accept a file-name as a search key.

Well Hidden

Things can get tricky if *./configure* fails to find a file that exists on your machine.

Listing 2: Failing to find an existing file

```
01 [andi@diabolos ogle-0.9.1]$ ./configure
02 [...]
03 checking for madvise... yes
04 checking for DVDDiscID in -ldvdread... no
05 checking for DVDOpen in -ldvdread... no
06 configure: error: Need libdvdread, install it or specify it's
   location
```

Listing 3: Error message with integrated help

```
01 [andi@diabolos lbreakout2-2.5beta-3]$ ./configure
02 [...]
03 checking for main in -lpng... yes
04 checking for sdl-config... no
05 checking for SDL - version >= 1.2.0... no
06 *** The sdl-config script installed by SDL could not be found
07 *** If SDL was installed in PREFIX, make sure PREFIX/bin is in
08 *** your path, or set the SDL_CONFIG environment variable to the
09 *** full path to sdl-config.
10 configure: error: lib SDL is needed
```

The reason for this might be that you compiled the required library yourself, or that you use a distro that stores files in unusual directories where *configure* will not look for them. Listing 2 provides an example of this.

In this example, *configure* cannot find *ldvdread*. This is not the name of a missing file – expand the *l* at the beginning of this name to *lib* and add *.so* to give you the missing name: *libdvdread.so*. Files that start with *lib* are program libraries that can be used by many different applications.

The missing *ldvdread* is part of the *libdvdread* package. We compiled this file ourselves and installed it below */usr/local/multimedia*. *locate libdvdread.so* tells us that the *libdvdread.so* library, which *configure* is now looking for, is stored below */usr/local/multimedia/lib*.

configure cannot find the file as */usr/local/multimedia* is not a standard directory for software installations. It can still make sense to use a directory like this, if you want to test an application before deleting it again, or if you are not the administrator of a machine, and do not have write privileges for the whole filesystem.

configure in Listing 2 actually points the way to a solution: *configure: error: Need libdvdread, install it or specify it's location*. As we know that the missing library is installed, it can only be a question of pointing in the right direction.

Calling *configure* with the *--help* flag (see Figure 4) outputs a list of parameters, including a short explanation of each one. This tells us that the *--prefix* parameter is used to specify a location when installing a program. This typically defaults to */usr/local*. As the command opens help in the *less* pager, you can type */dvdread* to search.

```
./configure -with-dvdread=/usr
/local/multimedia
```

is the right choice. *configure* expects the location in which you installed the application as the path.

Multiple Options

Some *configure* scripts provide precise instructions for resolving errors. In Listing 3, *Configure* cannot find *sdl-config*. First, check whether this file is installed. On a Mandrake system, the application is part of the *libSDL1.2-devel* package. If the file does not exist, install the package.

If you have a self-compiled version of *sdl-config* somewhere on your disk, *configure* provides a few useful tips in the lines starting with asterisks. *sdl-config* is a script that tells *configure* where the current **SDL** installation is located, and what **compiler flags** are needed in the *Makefile*. To allow *configure* to find the script, it needs to be located in the search *PATH* for executables. If *sdl-config* is located in */usr/local/games/bin*, enter the following command

```
export PATH=$PATH:/usr/local➥
/games/bin
```

to add the directory to the current path. The *$PATH* expression following the equals sign avoids overwriting the existing path. The new path is formed by concatenating the existing path and the */usr/local/games/bin* directory. The colon is the separating character. To permanently add */usr/local/games/bin* to the search path, add the command to the startup file for **Bash**, *.bash_profile*, in your */home* directory.

As an alternative, you can set the *SDL_CONFIG* environment variable, again using the *export* command:

```
export SDL_CONFIG=/usr/local➥
/games/bin/sdl-config
```

You need to supply the full path (in Listing 3: *full path to sdl-config*) to the *sdl-config* file.

Listing 4: *pkg-config* error message

```
01 [andi@diabolos gphoto2-2.1.1]$ ./configure
02 [...]
03 checking for libgphoto2 >= 2.1.1... Package libgphoto2 was not
04 found in the pkg-config search path.
05 Perhaps you should add the directory containing `libgphoto2.pc'
06 to the PKG_CONFIG_PATH environment variable
07 No package 'libgphoto2' found
08
09 configure: error: Library requirements (libgphoto2 >= 2.1.1) not
10 met; consider adjusting the PKG_CONFIG_PATH environment
11 variable if your libraries are in a nonstandard prefix so
12 pkg-config can find them.
```

Variables

Recently, there has been a tendency to use *pkg-config* to perform the tasks usually assigned to library-specific configuration scripts such as *sdl-config*. The

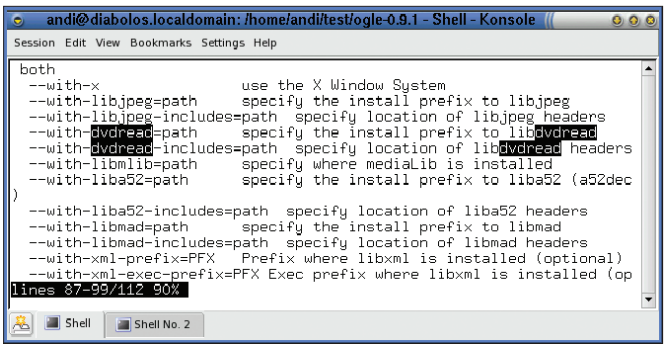

Figure 4: "./configure --help" outputs a list of all the parameters the configuration script knows.

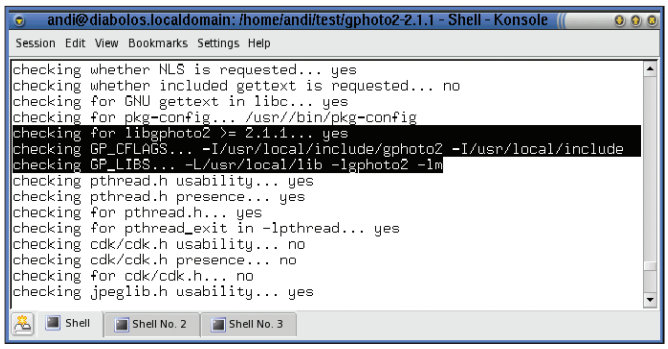

Figure 6: After defining the "PKG_CONFIG_PATH" correctly, "pkg-config" can provide the "configure" script with the right information.

script is a programming tool that discovers the location of a library and the parameters required to use the library to build programs.

The *pkg-config* tool parses simple text files with the *.pc* extension. These are normally located below the */usr/lib/pkconfig* directory. They provide information such as the installed library version, and the paths to the **include files** and libraries (see Figure 5 to the right).

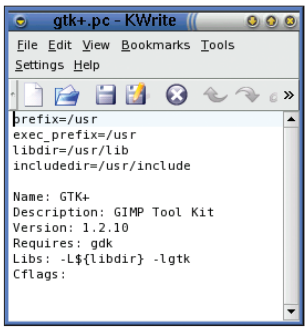

Figure 5: "pkg-config" parses the "gtk+.pc" file to learn all it needs to know about the installed GTK version.

If you compile a library yourself, for example *libgphoto* to access digital cameras, the files are installed below */usr/local* by default. "*make install*" copies *libgphoto2.pc* to */usr/local/lib/pkgconfig*. If you then attempt to build an application that needs this library to run, *./configure* will fail and output an error like the one which is shown in Listing 4.

On the Right Path

The *configure* script thinks that the installed version of *libgphoto2* is too old (*Library requirements (libgphoto2 >= 2.1.1) not met*). It comes to this conclusion as it was unable to find *libgphoto2.pc* in *pkg-config search path*. Obviously, *pkg-config* must have a search path in which it searches for *pc* files.

In Listing 4, *configure* helps out by telling us that we can use the *PKG_CONFIG_PATH* environment variable to ex- tend the path. It also tells us what directories that should be included in the path, i.e. the directory where *libgphoto2.pc* is stored (*Perhaps you should add the directory containing `libgphoto2.pc'*).

As previously mentioned, you can use the *export* command to define the environment variable. For example, the following one line command:

```
export PKG_CONFIG_PATH=/usr➥
/local/lib/pkgconfig:/usr➥
/lib/pkgconfig
```

builds a *pkg-config* search path using the */usr/local/lib/pkgconfig* directory, under which your self compiled *libgphoto2.pc* is stored, and */usr/lib/pkgconfig* is the default search path. The environment variable does not really need the standard path, as *pkg-config* will search this path whatever.

If you regularly compile programs, you might like to add a *PKG_CONFIG_PATH* definition to your *~/.bash_profile*. After doing so, the *configure* script that *pkg-config* calls should have no trouble finding your version of *libgphoto2* (see Figure 6).

More Help

Some *configure* errors are not covered by these examples. The script may not have an appropriate option, or the program author may have made a mistake. If you get stuck, there are still a few things you can try to solve the problem, before you give up. First, take a look at the *config.log* file in the source code directory. This is where the *configure* tool logs the commands it executes and their output.

If the logfile is no help, try feeding the error message to a search engine like Google. Newsgroups [2] in particular often have postings from users who have had the same problem, and found a solution. As a last resort, talk to the developer, an address should be included in *AUTHORS*. *BUGS* or *README* often include a description of the details the author will need to help you resolve the problem. ■

GLOSSARY

SDL: *"Simple Directmedia Layer" is a cross operating system library that contains functions for low-level access to sound cards, input devices, joysticks, 3D accelerator cards and the like.*

Compiler flags: *Options that tell the compiler where to find libraries and header files.*

Bash: *A command line interpreter (a.k.a the "shell"), just like "command.com" on DOS. Linux distributors use Bash as the standard shell, but there are numerous alternatives such as "tcsh", whose syntax is similar to C, or the extremely powerful zshell.*

Include files: *Files with the ".h" suffix, also known as "header files" that describe the interfaces of a library or application. They provide details on how to call a function. Programmers wanting to use functions of this type need to include the header files in their source code. Dev(el) packages are mainly made up of header files, which are needed to build applications.*

INFO

[1] Search engine for RPM and Debian packages: *http://www.rpmseek.com/*

[2] Usenet search on Google: *http://groups.google.com/*