What to do when make won't play ball.

Hand-Built

After successfully completing the configure script, users only need to negotiate the make and make install steps to build a new application. Although this sounds simple, the details can be a little tricky, when things first appear to go wrong with the make command. We guide you through the steps you need to complete, in order to successfully build an application.

BY ANDREA MÜLLER

Make and make install errors are strange things; they should not really exist. The actions that the *make* program performs while compiling and installing a software program are laid down in the Makefile, as created by the configure script. If configure discovers that a required library is located in the */usr/local/multimedia/lib* directory, this information will be stored in the Makefile as a compiler parameter.

Superficial Checks

The file should suit your system perfectly and *make* should run like a dream. If that does not happen, the source code for the program may be buggy. In this worst case scenario, you need some help from the developer.

However, you may be able to handle some issues yourself, such as problems with messy *configure* scripts that do not fulfill all the requirements for a successful *make* operation (see Listing 1).

If something goes wrong at the *make* stage, the program will typically issue several dozen error messages. The first message is the most important; all the others may result from it. In Listing 1 *make* fails because the compiler is unable to locate *mimepp.h*. This error occurs while make is compiling a file called *decodeRFC2047.cpp*. The error occurs in line *21*.

If the error message is obvious, as in *mimelib/mimepp.h: No such file or directory*, there is no need to investigate the source code. Simply use *urpmf mimepp.h* (Mandrake Linux), *pin mimepp.h* (Suse Linux), or a file search at *http://www.rpmseek.com/*, to locate the package with the missing file.

Mandrake Linux returns *kdenetwork-devel* as a result of the search. After installing the package, launch *make* again. This time, *gcc* should have no trouble finding *mimepp.h*, and *make* will not fail.

On rare occasions *make* may overlook a file, despite it being on your machine.

Absent-Minded Developers

If you feel confident enough to add something to the source code, you can easily remove errors like the one in Listing 2. Simply dissect the cryptic looking messages piece by piece. The compiler is just compiling *legacyimport.cpp*, as indicated at the beginning of each line. Line *143* refers to *KInputDialog*, but the compiler has no idea what this is (*undeclared (first use this function)*). *Undeclared* means that the *KInputDialog* class, which is being used in a function for the first time here, has not been defined anywhere that the compiler might be able to look. This means that a code segment required to build the program, or to be more precise, the statements for creating a window with a text entry line and an *OK* button, are missing.

Programmers use reusable classes to avoid rewriting the code needed to create objects of this complexity from scratch each time. If you need an object belonging to this class, you can simply call the *constructor*.

Listing 1: A missing check in the script

```
01 [andi@doomtrain kshowmail-3.1.0-pre1]$ make
02 [...]
03 decodeRFC2047.cpp:21:28: mimelib/mimepp.h: No such file or directory
04 decodeRFC2047.cpp: In function `QCString decodeQuotedPrintable(const
   QCString&)':
05 decodeRFC2047.cpp:40: error: `DwString' undeclared (first use this
   function)
06 [...]
07 make[2]: *** [decodeRFC2047.o] Error 1
08 make[2]: Leaving directory `/home/andi/test/kshowmail-3.1.0-
   pre1/kshowmail'
09 make[1]: *** [all-recursive] Error 1
10 make[1]: Leaving directory `/home/andi/test/kshowmail-3.1.0-pre1'
11 make: *** [all] Error2
```

GLOSSARY

Makefile: *This ruleset file contains all the commands and details required to build a program, for example the location of libraries and include files, the compiler commands, and the copying commands that move the finished application to the target directory.*

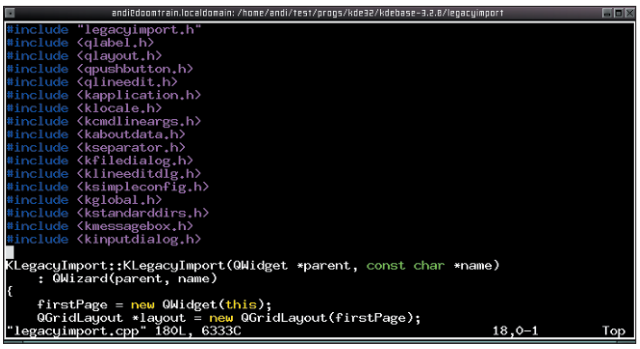

Figure 1: Adding an include line to "legacyimport.cpp".

Classes are program interfaces; C and C++ programs thus tend to place them in the header files of dev(el) packages, which typically end in an *.h* suffix. To allow the programs involved in compiling the source code to use the library functions, the developer adds include lines to the sources:

```
#include <kmessagebox.h>
```

tells the compiler to add the code stored in the *kmessagebox.h* file in one of the include directories. Include directories are passed to the compiler as makefile options, as in *-I/here/are/the/includes*. However, if the line states

```
#include "myinclude.h"
```

instead, that is, if the angled brackets are missing, the compiler expects the *myinclude.h* header in the same directory as the file it is compiling.

As you do not see a message about a missing file in Listing 2, the programmer has probably just forgotten to include the file which contains the class. To resolve this error, you first need to locate the file.

The *grep* command is useful for finding strings in files. Start off in the source code directory, or the *src* directory below it. *grep KInputDialog *.h* will search for the *KInputDialog* string in files with the *.h* suffix. This does not work in our example, as *KInputDialog* is a generic KDE class located in one of the system's include directories. These are typically */usr/include*, and */usr/local/include*, although KDE sometimes has */opt/kde3/include*.

```
grep -r KInputDialog /usr/➥
local/include/*
```

(*-r* for recursive) returns two matches on our lab system: *kinputdialog.h*, and *klineeditdlg.h*. The matching entry in the second file is just a comment, but the lines in *kinputdialog.h* look complicated enough to be a class. See Box 1: kinputdialog.h.

Also, the filename reminds you of a class, and that is a good sign, although it is in small letters. To find out if we guessed right, we can add the *legacyimport.cpp* below the other include lines (see Figure 1).

```
#include <kinputdialog.h>
```

Note that *make* no longer complains. This method will only work if the include file containing the required definition is located on your system.

Listing 2: A missing include line.

```
01 [andi@doomtrain]$ make
02 [...]
03 legacyimport.cpp: In member function `void
   KLegacyImport::finished()':
04 legacyimport.cpp:143: `KInputDialog' undeclared (first use this
   function)
05 legacyimport.cpp:143: (Each undeclared identifier is reported only
   once for each
06 function it appears in.)
07 legacyimport.cpp:143: parse error before `::' token
08 make: *** [legacyimport.o] Error 1
```

Box 1: kinputdialog.h

```
01 /usr/local/include/kinputdialog.h:class KInputDialog : public
   KDialogBase
02 /usr/local/include/kinputdialog.h:    KInputDialog( const QString
   &caption, const QString &label,
```

Installation with Obstacles

There are very few occasions where *make install* returns an error, and most of them occur when users attempt to install a program in a directory other than */usr/local*. If you want to place an application in */usr/local/test*, this may fail if the programmer forgets to add the lines required to create the target directory for the file copy operation to the *makefile*. If */usr/local/test/bin* does not exist, any attempt to copy a file to that directory is doomed to failure. The easiest way to remove an error of this kind is to use *mkdir* to create the target directories, whose names should be visible in the error messages.

Post-Install Blues

After building a new KDE application, you may still be in for a disappointment when you attempt to launch it. The program might not be able to locate its plug-ins, or it may have an empty tool bar, or talk to you in the wrong language (see Figure 2). By default, KDE applications only search for files in the directories where the KDE core applications and their cohorts reside. On Suse Linux, the KDE directory is */opt/kde3*, whereas Red Hat and Mandrake Linux opt for */usr*. If you build a KDE application, the icons, plug-ins and help files will be placed in */usr/local*, however.

In Web forums, unsuspecting seekers are often told to re-compile the application, and this time specify the KDE directory as the install target when calling *./configure*, e.g. *./configure --prefix=/opt/kde3*.

Although the application may run if you follow this advice, the answer is anything but perfect; in fact, it can be downright dangerous. */opt/kde3* on Suse, and the */usr* directory on Red Hat and Mandrake Linux are reserved for the package manager. Self-built programs

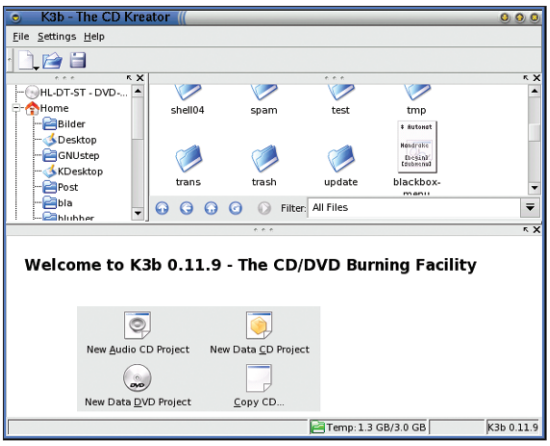

Figure 2: If you install K3b to "/usr/local" it will be unable to find its icons and plug-ins.

have no right to be there, as the package manager will not be able to remove them. The next time you update your distro, or KDE, these files can lead to an instable KDE. The self-built application may be long forgotten by this time, and even if you do remember, there is little chance of finding the culprit.

The correct approach is to tell your KDE programs that there is another data directory. You can use the *KDEDIRS* environment variable to do so:

```
export KDEDIRS=/usr/➥
local:/opt/kde3
```

points at both the Suse Linux directories: */usr/local*, for self-compiled software, and */opt/kde3* for KDE distribution components. The command for Red Hat and Mandrake Linux is *export KDEDIRS=/usr/local:/usr*.

Assuming that you have tuned your command line window in this way, KDE applications will look for data below */usr/local* in future. If you want to set this environment variable permanently, *.bash_profile* in your */home* directory is the right place to do so. Any commands you add to this file, are launched by bash when you log on to the system.

Suse Linux users have a bit more work to do, as the

```
unset KDEDIRS
```

command is called by */opt/kde3/bin/startkde*. This is the shell script that launches the KDE desktop. *unset* deletes the contents of the environment variable and is thus overrides the *export*. As the *~/.bash_profile* script runs before the desktop is launched, KDE will never know that there is another data directory.

To set the variable on Suse Linux, make sure that you are *root*, open */opt/kde3/bin/startkde* in your editor, and comment out (#) the line with *unset KDEDIRS*, to tell the shell not to run the rest of this line. You can easily remove the hash sign at a later stage. ■

THE AUTHOR

After nearly two-years as an independent journalist, Andrea Mueller works as a new editor for the Linux New Media AG.

When she is not taking care of articles, she looks beyond Linux and is involved with other operating systems like QNX and BeOS.