# Getting started with the *autotools*: a "Hello World" example

dominik.kern@student.tu-freiberg.de

November 3, 2006

Free Software comes as open-source packages, typically requiring compiling. According to the GNU-standards [1], the installation process should consist of three commands:

```
./configure
make
make install
```

This article describes the background of these commands and introduces the creation of such a package utilizing the *autotools*, particular: *autoconf* and *automake*.

## Contents

# 1 The Concept of Portability

The advantage of a higher programming language like C is that software developed on a certain system (instruction set, architecture and operating system) can be ported to any other system provided that there is a compiler. In contrast software written in machine-level language (assembly) is bounded to one system. Theoretically must only the source code be copied and compiled. In practice there are differences that must be worked around such as library and tool dependencies, environment variables, different directories and system specific restrictions.
Another point of distributing "open source" is the philosophy of free software [2], which states that software should be free and can be used-and-modified as long the new version is provided again as free software (freeware).
In order to understand the *autotools*, it is necessary to know how Makefiles work. This explanation is sketched out in the next section. Skip it, if you know how to write a Makefile. The subsequent sections describe the software distribution process and the application of the *autotools* in order to build a portable package for a "Hello World" program.

# 2 Makefiles

Makefiles control the compilation, installation and deinstallation of a software project. A one-file-program can be compiled easily from the command line. For bigger projects it is recommended to use the *make-build system* [3]. The advantages are as follows:

- One invocation for several files and parameters.
- The compilation can be reproduced by the Makefile.
- Dependencies between files are automatically resolved, saving compilation time since only updated files are compiled, while preventing mistakes which can occur due to forgotten dependencies.

## 2.1 The `make` concept

The `make` concept may initially seem a bit abstract, but it should be clarified by the following example. A Makefile consists of rules of the form:

```
<target>:<prerequisites>
        <commands>
```

Be aware that commands are written on separate lines and that tabs are placed in front of them. According to the coding standards [1] the following standard targets must be defined:

**all** Compiles the sources in the source directory without affecting the system

**install** Installs the compiled files into the system i.e. copying into the system directories, usually run by the appropriate user, who has the necessary permissions

**clean** Removes all generated files

**uninstall** Removes the installed files from the system, usually run by the appropriate user

Further targets might be useful such as **check, install-check, distclean** and there can be user defined targets such as intermediate targets. Invoking `make` without any parameter attempts to build the first target in the Makefile. Typically the first target is "all".
Prerequisites are required files or intermediate targets; they are checked for existence or executed respectively. If their dependencies can not be resolved, then the target fails (i.e. it cannot be built).
Each target follows commands, which produce it by compiler/program invocations or by calls of other targets.
In order to make Makefiles more flexible, it is good practice to define variables at the beginning of a Makefile for frequently used file and directory names.

## 2.2 A basic example Makefile for the "Hello World" program

Like object oriented programming, Makefiles utilize their full benefits in big projects. The following example is only for demonstration. A "Hello World" program is distributed in three source files: *main.cxx, hello.cxx, world.cxx*. These files should be compiled into object files and linked together into an executable. The source code can be found in the last section (5.1). The Makefile for this task could be as follows:

---

```
# Makefile for a "Hello World" program
# all:        compile the sources and create application "helloworld"
# clean:      delete all object files and the executable
# install:    copy executable "helloworld" to /usr/local/bin
# uninstall:  delete executable "helloworld" from /usr/local/bin

NAME=helloworld
CXX=g++
BINDIR=/usr/local/bin

all: $(NAME)  # create an executable $(NAME) since it is a prerequisite

clean:
        rm -f $(NAME)   # delete executable
        rm -f hello.o world.o main.o   # delete object files

install: $(NAME)
        cp $(NAME) $(BINDIR)    # copy executable in directory
                                # for user applications (bindir)

uninstall:
        rm -f $(BINDIR)/$(NAME) # delete executable from directory
                                # for user applications (bindir)

$(NAME): hello.o world.o main.o
        $(CXX) -o $(NAME) hello.o world.o main.o
 # link object files to executable
```

```
hello.o: hello.cxx hello.hxx
        $(CXX) -c hello.cxx
 # compile source file to object file

world.o: world.cxx  world.hxx
        $(CXX) -c world.cxx
 # compile source file to object file

main.o: main.cxx
        $(CXX) -c main.cxx
 # compile source file to object file
```

*Makefile*

## 2.3 An advanced example Makefile for the "Hello World" program

Instead of writing a rule for each object file, it would be more effective to write a general rule for creating object files from source files. This is done using suffix-rules. Targets and prerequisites can be defined as patterns, e.g. "`%.o`" for all files with the suffix ".o". In order to process files without declaring their names you need to utilize "automatic variables". These variables are evaluated by the *make build system* for each file name where the rule applies. Very useful automatic variables are `$@` for the target's file name and `$<` for the first prerequisite's name. In the example Makefile the two rules for building the object files *hello.o* and *world.o* would match the pattern:

```
# Compile all *.cxx files (*.hxx required too) into *.o files:
%.o: %.cxx %.hxx
        $(CXX) -c $< -o $@
# would result in commands like "g++ -c hello.cxx -o hello.o"
# since $< is replaced by the first prerequisite and $@ by
# the target name (automatic variables)
```

Further the list of object files could be stored in a variable.

```
OBJECTS= hello.o world.o main.o
```

Such a Makefile is elegant; however, it is static, i.e. it does not adapt to different systems (e.g. another *bindir*) and does not check if the *g++* compiler even exists. The next section describes how software is universally distributed to other users.

# 3 Software distribution process

In order to transfer software fast and easily, it is compressed into archives (e.g. **.tar.gz*). Usually after unpacking, there is a directory which contains the documentation files, subdirectories containing the source code, and for portability the *configure* script and templates like *Makefile.in* and *config.h.in*.
Running `./configure` examines the system for all the requirements (programs, libraries, etc.) and converts the templates (**.in*) into the final files (*Makefile, config.h,...* ), which are fitted to the target system. A *config.log* file is created for

debugging. After the successful execution of `./configure`, the Makefile is ready to compile the package by entering `make`. The compilation "should" work without any problems, and afterwards, the package can be installed by entering `make install`. The division in these three commands is useful, because neither compiling source code without meeting the requirements nor installing improperly compiled binaries makes sense. Except for the requirements of the package itself, the target system only needs a shell interpreter and the *make build system* [3]. The creation of the *configure*-script and the templates (**.in*) can be done very efficiently and comfortably with the *autotools* which are explained next. Figure 1 illustrates the installation on the target system and anticipates the use of the *autotools*.

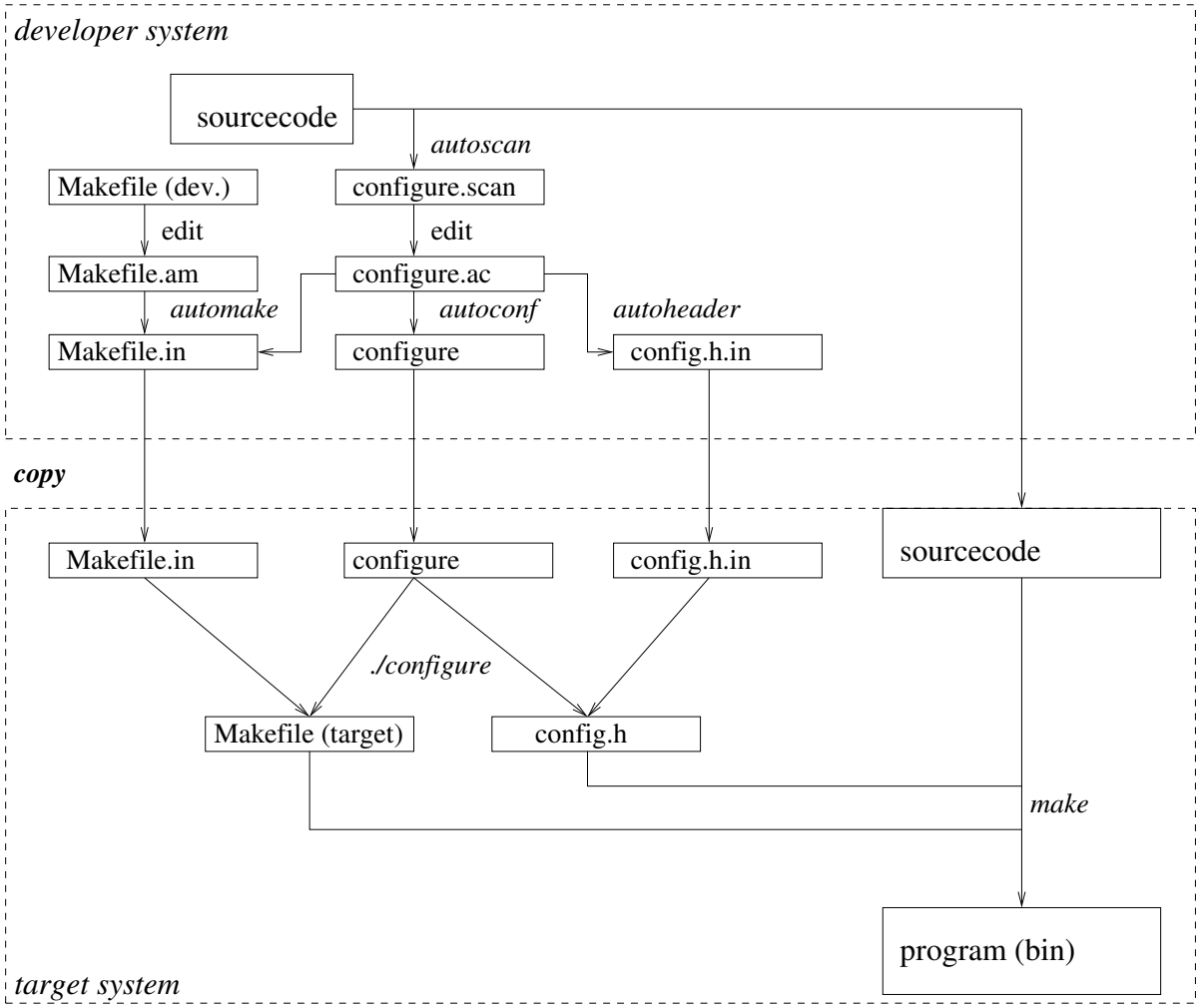


Figure 1: Creating and distributing portable software using the *autotools*

# 4 Using the *autotools*

Assuming your source code is done, the procedure for each of the *autotools* is similar. An input file needs to be written, and then the according tool is run. The *autotools* are a tool-chain, meaning that several tools can be combined or used separately. The tool-chain concept is powerful; however, you need to get an overview about the dependencies and where to start. To give an outline for this section, the tools are introduced step by step. At first it will be described how to use *autoconf* in order to generate a *configure*-script, next using *autoheader* additionally to have a *config.h.in*

template generated, followed by the addition of *automake* in order to "automate" the generation of a *Makefile.in* template. The next section describes how to apply the *autoconf/automake* combination to packages containing subdirectories. Subsequent the complete process of creating portable packages is summarized. Finally, some other useful tools are mentioned briefly to demonstrate further possibilities.
As an example a "Hello World" program uses the following files:
*main.cxx*, *hello.cxx*, *hello.hxx*, *world.cxx*, *world.hxx*. It is assumed that the *autotools* are installed. They are part of many distributions and integrated development environments (IDE). The latest versions can be downloaded from *www.gnu.org*. They require the *M4*-macro-processor and a *Perl*-interpreter.

## 4.1 *autoconf* only

*Autoconf* is launched by entering `autoconf`. It generates the *configure* script from the input file *configure.ac* (deprecated name *configure.in*). This file contains the macros (*M4*) about what to do on the target system. This file is also of relevance for other *autotools*. The *autoconf* package comes with some other useful tools, which will be included in the following description: *autoscan* (section 4.1), *autoheader* (section 4.2), *autoreconf* (section 4.5), and *autoupdate* (section 4.5). Running `autoscan` in the top level directory of the package creates a file *configure.scan*, a proposal for *configure.ac*. For our example it would be something like:

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([world.cxx])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CXX

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

*configure.scan*

This file is slightly modified to match our needs and saved as *configure.ac*. We enter the package name, version no., comments, and disable unused features:

```
#                                                -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59) ## prerequisite autoconf version in order to create
                ## the configure script, older version might fail
                ## due to unknown macros/names

AC_INIT(helloworld, 0.0.1, dev@helloworld.org) # package name and no.
                                               # and email

AC_CONFIG_SRCDIR([world.cxx]) ## representative source file
##AC_CONFIG_HEADER([config.h]) ## needed for autoheader, not used yet

# Checks for programs.
AC_PROG_CXX ## Check for C++ compiler

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_CONFIG_FILES([Makefile]) ## Create Makefile
AC_OUTPUT
```

*configure.ac*

Further macros could be added to check for certain programs, libraries, and functions, for example:

```
AC_PROG_RANLIB   # checking for ranlib
AC_PROG_F77      # checking for F77 fortran-compiler

AC_CHECK_LIB([my_lib],[main])   # checking for library "my_lib"
                                # containing function "main"

AC_CHECK_FUNCS([floor pow sqrt])  # checking for the floor, power-of
                                  # and square-root functions
```

There is much more to check for. With many macros coincident output variables e.g. `CXX` for the C++ compiler found. Macros and output variables are systematically named. All *autoconf* macros begin with `AC_` and the next word refines the function (`AC_CHECK_..` , `AC_PROG_..`). There is an index only for macros and output variables in the *autoconf* manual [4]. You can also write your own macros or utilize general purpose macros such as `AC_CHECK_PROG`. Here is an example regarding how to check for a certain program:

```
AC_CHECK_PROG( MYPROG, [myprog], myprog, no )
if test $MYPROG = no; then
    AC_MSG_ERROR([myprog is not installed.])
fi
```

The parameters for `AC_CHECK_PROG( , , , )` are:

1. Name of the output variable which contains the result
2. Name of the executable to search for
3. Value of the output variable if program was found
4. Value of the output variable if program was not found

The following lines evaluate the check. In case the check was negative (Value of `MYPROG` is `no`), `AC_MSG_ERROR` is called. This macro prints the error message and aborts the `./configure` process. An alternative is `AC_MSG_WARN`, which prints a warning message and continues the `./configure` process .
Output variables can be used in the *Makefile.in* template. Running `./configure` on the target system produces the Makefile by substitution of the output variables in the *Makefile.in* template. For our "Hello World" program it would look like the static Makefile in section 2.2 except the two lines which assign the C++ compiler found and the binary directory to our own Makefile variables:

---

```
# Makefile template for a "Hello World" program

NAME=helloworld
OBJECTS= hello.o world.o main.o

# use ./configure results, .i.e. macro output variables:
CXX=@CXX@
BINDIR=@bindir@

all:     $(NAME)
clean:
        rm -f $(NAME)
        rm -f $(OBJECTS)
install: $(NAME)
        cp $(NAME) $(BINDIR)
uninstall:
        rm -f $(BINDIR)/$(NAME)
$(NAME): $(OBJECTS)
        $(CXX) -o $(NAME) $(OBJECTS)

# Compile all *.cxx files (*.hxx required too) into *.o files:
%.o: %.cxx %.hxx
        $(CXX) -c $< -o $@
# would result in commands like "g++ -c hello.cxx -o hello.o"
# since $< is replaced by the first prerequisite and $@ by
# the target name (automatic variables)
```

```
main.o: main.cxx
        $(CXX) -c $< -o $@
# this rule is listed additionally since there is no prerequisite main.hxx
```

*Makefile.in*

`CXX` is an output variable of the `AC_CHECK_CXX` macro and `bindir` is an output variable of `AC_INIT` which also detects `includedir, srcdir, libdir,` and others. The *autoconf* output variables inclosed by @ will be substituted by the found values (e.g. `CXX` by `g++`). Our Makefile variables could be named anything e.g. `COMPILER`, but it is best to stick to standardized names.

## 4.2 *autoconf* and *autoheader*

You can take advantage of the things `./configure` detects in your programs. Including *config.h* enables the use of the `./configure` results. The file *config.h* contains a bunch of preprocessor definitions for system specific features which are detected by `./configure`. In order to enable the generation of *config.h* from the template, add the following line to *configure.ac*:

```
AC_CONFIG_HEADER([config.h])
```

In order to create the template file *config.h.in*, just run `autoheader`. It is useful to have a look in the template *config.h.in* to see what is figured and what the preprocessor definitions are.
For instance you can use `PACKAGE_STRING` in your program. The constant is defined since `AC_INIT` is in *configure.ac*. Our "Hello World" program uses it:

```
#include <config.h>  // make configure results available
...
cout<<PACKAGE_STRING; // print Package name and Version no.
```

*main.cxx*

## 4.3 *autoconf* and *automake*

This combination is very powerful. You will not only get the *configure* script generated but also the template *Makefile.in*. *Automake* interacts with *autoconf* and cannot be used without it. The file *configure.ac* must include the line:

```
AM_INIT_AUTOMAKE
```

*Automake* is launched with the command `automake` and controlled by the input file *Makefile.am*. This file contains "what needs to be built from which sources". Our "Hello World" is a binary program built from the source files *hello.cxx, world.cxx,* and *main.cxx*:

```
bin_PROGRAMS=helloworld
helloworld_SOURCES=hello.cxx hello.hxx world.cxx world.hxx main.cxx
```

*Makefile.am*

In order to pass flags there can be variables defined such as `helloworld_CXXFLAGS` for the C++ compiler, `helloworld_CPPFLAGS` for the C preprocessor, or for linking with libraries there would be variables like `helloworld_LDADD`.

## 4.4 Processing subdirectories

There are two possibilities for processing subdirectories. Two existing packages can be combined to a nested package. The top packages calls the inner package (assumed directory name ‘‘SUBDIR1’’) by adding the following lines to the input files in the top level directory:

```
configure.ac: AC_CONFIG_SUBDIRS([SUBDIR1])

Makefile.am: SUBDIRS=SUBDIR1
```

There is a detailed example (“arm/hand”) in the *automake* manual [5]. This way is clear since each package can be created separately. Thus the modularity is preserved. On the other hand several *configure* scripts will be run, one for each directory, repeating previously completed checks, like checking for the C++ compiler repeatedly. Also the auxiliary scripts such as *install-sh* will be generated for each directory. Another possibility is to create one *configure* script for several directories (e.g. there is a library which is built first, and then linked with the program). The concept of libraries is sketched out in section 5.2. Splitting the “Hello World” program into a static library and a main program would end up in the following structure:

```
package_directory
|
|--include
|  ’--hello.hxx world.hxx
|
|--lib
|  |--hello.cxx world.cxx
|  ‘--Makefile.am
|
|--src
|  |--main.cxx
|  ‘--Makefile.am
|
|--configure.ac: ..AC_CONFIG_FILES([lib/Makefile src/Makefile Makefile])..
|--Makefile.am:  SUBDIRS=lib src  # processed in this order !
‘--README, NEWS, INSTALL, ChangeLog, AUTHORS, COPYING
```

The `configure` command must check for all requirements, for the libraries in *./lib* as well as for the program in *./src*. If the checks succeed it creates a Makefile in the top level directory and in each of the subdirectories. Building a library requires an additional tool, which generates an index to an archive, e.g.: `ranlib`.

---

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59)
AC_INIT(helloworld, 0.0.1, dev@helloworld.org)
AM_INIT_AUTOMAKE  ## Automake is used
AC_CONFIG_SRCDIR([src/main.cxx])  ## representative source file
AC_CONFIG_HEADER([config.h])  ## Autoheader is used
```

```
# Checks for programs.
AC_PROG_CXX
AC_PROG_RANLIB  ## needed to build a library

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

# creating top level and subdirectory Makefiles:
AC_CONFIG_FILES([Makefile
                 src/Makefile
                 lib/Makefile])
AC_OUTPUT
```

*configure.ac* (top level)

The top level Makefile is pointing to the subdirectories. The subdirectories are processed in the given order:

```
SUBDIRS=lib src ## processing subdirs in given order
```

*Makefile.am* (top level)

In the directory *lib* a static library is built out of the sources *hello.cxx* and *world.cxx*:

```
noinst_LIBRARIES=libhw.a  ## static library which is not to be installed
libhw_a_SOURCES=hello.cxx hello.hxx world.cxx world.hxx
libhw_a_CXXFLAGS=-I../include  ## add path to headerfiles
```

*lib/Makefile.am*

In the *src* directory the main program is built and linked with the previously built static library into one executable program:

```
bin_PROGRAMS=helloworld
helloworld_SOURCES=main.cxx
helloworld_CXXFLAGS= -I../include  ## add path to headerfiles
helloworld_LDADD=../lib/libhw.a  ## link with static library
```

*src/Makefile.am*

## 4.5 Summary of steps to create a portable package

The complete process of creating portable packages is summarized in this section. Assuming the source is done, and there are documentation files according to the coding standards [1]:

**README** the first look, contains a summary about the content and a reference to the file *INSTALL*

**INSTALL** the installation instructions

**COPYING** the copying conditions

**NEWS** all changes to the program concerning the user such as new functions

**ChangeLog** all changes to the source in order to find bugs quickly

**AUTHORS** programmers name and contact information

For testing it is enough to run `touch NEWS README ChangeLog AUTHORS` and have *INSTALL* and *COPYING* automatically added by *automake*. Before distributing a package, fill in the correct content!
Starting from the top level directory of the package, the sequence of steps is as follows:

**autoscan** (optional) Generates *configure.scan*, a proposal for *configure.ac* derived from the source code

**aclocal** Places *M4* macro definitions needed by *autoconf* into a single file. The `aclocal` command first scans for macro definitions in **.m4* files in its default directory (*/usr/share/aclocal* on some systems) and in the file *acinclude.m4*. It next scans for macros used in the *configure.ac* file. It generates an *aclocal.m4* file that contains definitions of all *M4* macros required by *autoconf* and a directory *autom4te.cache*.

**edit configure.ac** Adjust the generated proposal *configure.scan* or write the file from scratch

**autoconf** Generates the *configure* script

**autoheader** (optional) Generates the template file *config.h.in*

**create Makefile.in** Either by hand from the static Makefile or edit *Makefile.am* and run `automake -a` The parameter "a" adds missing files and scripts.

Except `autoscan`, these steps are repeated after every change of the input files. In order to "automate" the reconfiguration they are compressed into one powerful command:
`autoreconf -vfi`.
It rebuilds *configure* and templates. The parameters are as follows:

**v** verbose, to get information about the reconfiguration process

**f** force, to replace existing files

**i** install, to copy missing files into the package

With newer versions of *autoconf* there were changes to the *autoconf* macro naming scheme. To "modernize" an old file *configure.ac* there is the command `autoupdate`. It substitutes obsolete macro names to the current names.

For further information check the manuals [3] [4] [5]. There are very helpful indices for macros and variables.
Many packages often still contain the input files (*configure.ac Makefile.am*). Read them! Learn from others. Maybe you can fix errors when `./configure` or `make` fails for downloaded software.

If software projects are under version control, such as CVS or Subversion, there are no files in the repository, which could be generated automatically. Thus there are only the input files (*configure.ac Makefile.am*). In order to organize the calls of the *autotools* there is a script typically named *autogen.sh*. Unfortunaly this name is similar to the name of an existing software tool for automatic text generation "autogen". There is no relation between *autogen* and *autogen.sh*.

## 4.6 Keywords for further information

Now you have learned the basic *autotools* in order to create a portable package. There are more useful tools for helping you to "automate" standard procedures in software development. This section gives some advice for further information. There is a bunch of additional useful tools in connection with the *autotools*. Sometimes it is helpful enough to know that a tool exists. Here are some keywords that might lead you to a fast solution.

**autoqt** Checks for QT [6]. This is a frequent task in many applications, particularly with graphical user interface (GUI). There is a macro `gw_CHECK_QT` which checks for QT, detects several parameters such as version no., *QTDIR*, paths to *uic, moc* and provides them as variables. It can be called from *configure.ac* presuming the file *gwqt.m4* was copied to the *autoconf* directory. This file is available from the autoqt-project,
*http://autoqt.sourceforge.net*

**gettext** Prepares your programs easily for internationalization (abbreviated "i18n"). All strings in the source that should be prepared for translation, usually the output for the user, need to be put in a macro. This macro extracts the strings to a file (**.pot*) which can be translated. Using `AM_GNU_GETTEXT` in *configure.ac* supports gettext. It evaluates the local language setting and chooses the translation accordingly,
*http://www.gnu.org/software/gettext/gettext.html*

**kdoc** Generates documentation from the source code and its comments,
*http://sirtaj.net/projects/kdoc/*

**libtool** Supports programming and distributing of libraries. Run `libtoolize` on the developer system to create a shell script named *libttool*. On the target system this script manages to provide the dynamic library, it might even simulate a dynamic library using static libraries,
*http://www.gnu.org/software/libtool/*

## 4.7 Conclusion

Using the *autotools* from scratch is as easy as writing Makefiles. I hope that I have encouraged you to utilize the *autotools* for your future projects. At any time your programs are portable and you can share them with others and in addition the

*autotools* are supported by many integrated development environments (IDE), e.g. *KDEVELOP*.
The *M4*-macro-processor is aged by now, maybe it will be substituted soon, but the idea will continue. The drawback of alternative approaches such as *cmake* [7] is that it needs to be installed on the target system as well.

# 5 Background information

## 5.1 Source code of the "Hello World" example

```
#include "hello.hxx"
#include "world.hxx"
#include "config.h" // make configure results available

int main()
{
hello first_word;
world second_word;

std::cout<<PACKAGE_STRING; /* use the preprocessor definitions
from config.h */

first_word.print();
second_word.print();

return 0;
}
```

*main.cxx*

```
#include <iostream>

#ifndef HELLO_HXX
#define HELLO_HXX

class hello{
public:
        void print();
};

#endif
```

*hello.hxx*

```
#include "hello.hxx"

void hello::print()
{
        std::cout<<" Hello ";
}
```

*hello.cxx*

The class "world" (*world.hxx*, *world.cxx*) is identical with the class "hello" except the class name and the output string. All example files can be found at:
*http://kern-81.homepage.t-online.de/autotools_examples.tgz*

## 5.2 Library concept

Libraries provide external functionality; the linking is done in order to build the executable. The header files of the libraries are necessary for the compilation, circumstantiating the preprocessing. They provide the function names and parameter declarations.

There are two kinds of libraries: static libraries (**.a*) and dynamic libraries (**.so*). Static libraries are archived object files. They are merged with the compiled program during linking. Static libraries are only needed once for linking. The built executable contains all the code so there will never be a dependency from a program to a static library. This make things lucid; however, the programs bulk up.

Dynamic libraries, a.k.a. "shared objects", are accessed during the run time of a program. These libraries are usually located in the *libdir* (e.g. */usr/lib*). The executable contains references instead of merged code. The dependencies from dynamic libraries can be found using the command `ldd <programfile>`. There are differences in the handling of dynamic libraries from one system to another. Some systems even do not support dynamic libraries.

# References


[1] GNU coding standards, online-manual *http://www.gnu.org/prep/standards*

[2] Philosophy of the GNU Project *http://www.gnu.org/philosophy*

[3] GNU make 3.80, online-manual *http://www.gnu.org/software/make*

[4] GNU autoconf 2.59, online-manual *http://www.gnu.org/software/autoconf*

[5] GNU automake 1.96, online-manual *http://www.gnu.org/software/automake*

[6] QT, class library for cross-platform application development *http://www.trolltech.com*

[7] CMake, the Cross-platform Make *http://www.cmake.org*

[8] Matthias Kupfer, interview at "Chemnitzer Linux-Tage 2006"