# The DUNE Buildsystem HOWTO

Christian Engwer*

March 1 2009

*Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg,
Im Neuenheimer Feld 368, D-69120 Heidelberg, Germany

http://www.dune-project.org/

## Contents

# 1 Getting started

**TODO: How do I build the grid howto?**

# 2 Creating a new DUNE project

From a build system point of view there is no difference between a DUNE application and a DUNE module.

DUNE modules are packages that offer a certain functionality that can be used by DUNE applications. Therefore DUNE modules offer libraries and/or header files. A DUNE module needs to comply with certain rules (see 3).

Creating a new DUNE project has been covered in detail in 2.1 using duneproject to take work off of the user. This is also the recommended way to start a new project. If for whatever reasons you do not wish to use duneproject here is the bare minimum you have to provide in order to create a new project:

- a dune.module file
  Usually you will only need to specify the parameters Module and Depends.
- *Note:* an dune-autogen script is *not* needed any more!
- a basic m4 file
  You need to provide two macros *MODULE*_CHECKS and *MODULE*_CHECK_MODULE (see 5.5).
- a configure.ac file
  Have look at the configure.ac in dune-grid for example. The most important part is the call to DUNE_CHECK_ALL which runs all checks needed for a DUNE module, plus the checks for the dependencies.

## 2.1 Configuring new DUNE module using duneproject

This section tells you how to begin working with DUNE without explaining any further details. For a closer look on duneproject, see section 2.

Once you have downloaded all the DUNE modules you are interested in, you probably wonder "How do I start working with DUNE?" It is quite easy. Let us assume you have a terminal open and are inside a directory containing some DUNE modules. Let us say

```
ls -l
```

produces something like:

```
dune-common/
dune-grid/
config.opts
```

There is no difference between a DUNE module you have downloaded from the web and modules you created yourself. dunecontrol takes care of configuring your project and creating the correct Makefiles (so you can easily link and use all the other DUNE modules). It can be done by calling

```
./dune-common/bin/duneproject
```

*Note:* In case you are using the unstable version DUNE you should be aware that the build system may change, just like the source code. Therefore it might be that duneproject is not up to date with the latest changes.

After calling duneproject, you have to provide a name for your project (without whitespace), e.g., dune-foo. The prefix dune- is considered good practice, but it is not mandatory. You are then asked to provide a list of all modules the new project should depend on (this will be something like dune-common dune-grid, etc.). At last, you should provide the version of your project (e.g., 0.1) and your email address. duneproject now creates your new project which is a folder with the name of your project, containing some files needed in order to work with DUNE. In our example,

```
ls -l dune-foo/
```

should produce something like

```
configure.ac
dune.module
Makefile.am
README
src
 --> dune-foo.cc
doc
```

You can now call dunecontrol for your new project, as you would for any other DUNE module. If you have a config.opts file configured to your needs (see e.g. the "Installation Notes" on http://www.dune-project.org), a simple call of

```
./dune-common/bin/dunecontrol --module=dune-foo --opts=config.opts all
```

should call dune-autogen, configure and make for your project and all modules your project depends on first.

**Remark 2.1** *Always call dunecontrol from the directory containing dune-common.*

You can now simply run

```
./dune-foo/src/dune-foo
```

which should produce something like

```
Hello World! This is dune-foo.
This is a sequential program.
```

If you want your DUNE module to be usable by other people your design should follow a certain structure. A good way to indicate that your module is set up like the other DUNE modules is by naming it with the prefix dune-. Since your module should be concerned with a certain topic, you should give it a meaningful name (e.g. dune-grid is about grids). You will also see that there are subfolders doc/, foo/ and src/ in dune-foo/. foo/ will contain any headers that are of interest to other users (like the subfolder common/ in dune-common, grid/ in dune-grid, etc.). Other users will have to include those files if they want to work with them. Let's say your project provides some interface implementation in a file foo.hh. duneproject already put this an example file into the subfolder dune/foo/.

```
dune-foo/
-> configure.ac
-> doc/
```

```
   -> doxygen/
      -> Doxylocal
      -> Makefile.am
   -> Makefile.am
-> dune.module
-> dune/
   -> foo/
      -> foo.hh
      -> Makefile.am
   -> Makefile.am
-> Makefile.am
-> README
-> src/
   -> dune_foo.cc
```

After running

```
make doc
```

in `dune-foo` you should now find a `html doxygen` documentation in `dune-foo/doc/doxygen/html/index.html`.

## 3 Dune module guidelines

A `DUNE` module should comply with the following rules:

- Documentation is located under `doc/` and gets web-installed under `BASEDIR/doc/`.
- `automake` includes are located in `dune-common`. To use them, you will have to make a symbolic link to `dune-common/am/` (see 5.1.2). The symlink creation should be handled by the `dune-autogen` (see 5.4).
- The `am/` directory does not get included in the tarball.
- Additional configure tests are located in the `m4/` directory. You should at least provide the macros *MODULE*`_CHECKS` and *MODULE*`_CHECK_MODULE`, in order to setup and find your module (see 5.5).
- Header files that can be used by other `DUNE` modules should be accessible via `#include <dune/foo/bar.hh>`. In order to work with a freshly checkout version of your module you will usually need to create a local symbolic link `dune -> `*module-directory*`/`. This link gets created by the `DUNE_SYMLINK` command in your `configure.ac`. When running `make install` all header files should be installed into *prefix*`/include/dune/`.

## 4 The Structure of `DUNE`

`DUNE` consists of several independent modules:

- `dune-common`
- `dune-grid`
- `dune-istl`
- `dune-grid-howto`

- `dune-grid-dev-howto`

Single modules can depend on other modules and so the `DUNE` modules form a dependency graph. The build system has to track and resolve these inter-module dependencies.

The build system is structured as follows:

- Each module is built using the GNU AutoTools.
- Each module has a set of modules it depends on, these modules have to be built before building the module itself.
- Each module has a file `dune.module` which holds dependencies and other information regarding the module.
- The modules can be built in the appropriate order using the `dunecontrol` script (shipped with `dune-common`)

The reasons to use the GNU AutoTools for `DUNE` were the following

- We need platform independent build.
- Enabling or disabling of certain features depending on features present on the system.
- Creations of libraries on all platforms.
- Easy creation of portable but flexible Makefiles.

The reasons to add the `dunecontrol` script and the `dune.module` description files were

- One tool to setup all modules (the AutoTools can only work on one module).
- Automatic dependency tracking.
- Automatic collection of command-line parameters (`configure` needs special command-line parameters for all modules it uses)

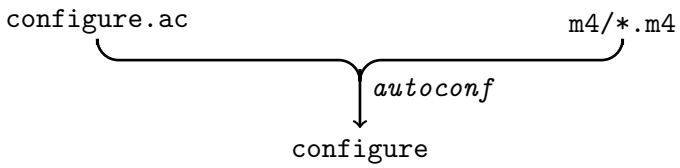
## 5 Building Single Modules Using the GNU AutoTools

Software is generally developed to be used on multiple platforms. Since each of these platforms has different compilers, different header files, there is a need to write makefiles and build scripts that work on a variety of platforms. The Free Software Foundation (FSF), faced with this problem, devised a set of tools to generate makefiles and build scripts that work on a variety of platforms. These are the GNU AutoTools. If you have downloaded and built any GNU software from source, you are familiar with the `configure` script. The `configure` script runs a series of tests to get information about your machine.

The autotools simplify the generation of portable Makefiles and configure scripts.

### autoconf

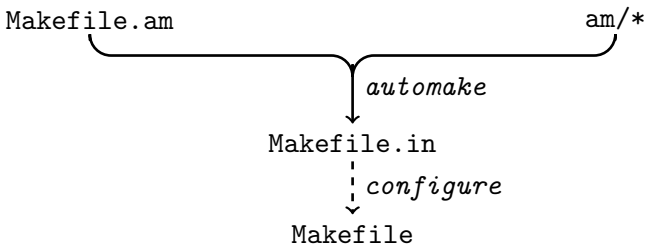
`autoconf` is used to create the `configure` script. `configure` is created from `configure.ac`, using a set of `m4` files.




How to write a `configure.ac` for `DUNE` is described in Sec. 5.2.

**automake**

`automake` is used to create the `Makefile.in` files (needed for `configure`) from `Makefile.am` files, using a set of include files located in a directory called `am`. These include files provide additional features not provided by the standard `automake` (see Sec. 5.1.2). The `am` directory is in the `dune-common` module and each module intending to use one of these includes has to have a symlink `am` that points to `dune-common/am`. This link is usually created by `dune-autogen` (see Sec. 5.4).




Information on writing a `Makefile.am` is described in 5.1

**libtool**

`libtool` is a wrapper around the compiler and linker. It offers a generic interface for creating static and shared libraries, regardless of the platform it is running on.

`libtool` hides all the platform specific aspects of library creation and library usage. When linking a library or an executable you (or `automake`) can call the compiler via `libtool`. `libtool` will then take care of

- platform specific command-line parameters for the linker,
- library dependencies.

**configure**

`configure` will run the set of tests specified in your `configure.ac`. Using the results of these tests configure can check that all necessary features (libraries, programs, etc.) are present and can activate and deactivate certain features of the module depending on what is available on your system.

For example `configure` in `dune-grid` will search for the ALUGrid library and enable or disable `Dune::ALU3dGrid`. This is done by writing a preprocessor macro `#define HAVE_ALUGRID` in the `config.h` header file. A header file can then use an `#ifdef` statement to disable parts of the code that do not work without a certain feature. This can be used in the applications as well as in the headers of a `DUNE` module.

The `config.h` file is created by `configure` from a `config.h.in` file, which is automatically created from the list of tests used in the `configure.ac`.

## 5.1 Makefile.am

### 5.1.1 Overview

Let's start off with a simple program *hello* built from `hello.c`. As `automake` is designed to build and install a package it needs to know

- what programs it should build,
- where to put them when installing,
- which sources to use.

The core of a `Makefile.am` thus looks like this:

```
noinst_PROGRAMS = hello
hello_SOURCES = hello.c
```

This would build *hello* but not install it when `make install` is called. Using `bin_PROGRAMS` instead of `noinst_PROGRAMS` would install the *hello*-binary into a *`prefix`*`/bin` directory.

Building more programs with several source files works like this

```
noinst_PROGRAMS = hello bye

hello_SOURCES = common.c common.h hello.c
bye_SOURCES = common.c common.h bye.c parser.y lexer.l
```

`automake` has more integrated rules than the standard make, the example above would automatically use yacc/lex to create `parser.c/lexer.c` and build them into the *bye* binary.

Make-Variables may be defined and used as usual:

```
noinst_PROGRAMS = hello bye

COMMON = common.c common.h

hello_SOURCES = $(COMMON) hello.c
bye_SOURCES = $(COMMON) bye.c parser.y lexer.l
```

Even normal make-rules may be used in a `Makefile.am`.

#### Using flags

Compiler/linker/preprocessor-flags can be set either globally:

```
noinst_PROGRAMS = hello bye

AM_CPPFLAGS = -DDEBUG

hello_SOURCES = hello.c
bye_SOURCES = bye.c
```

or locally:

```
noinst_PROGRAMS = hello bye

hello_SOURCES = hello.c
hello_CPPFLAGS = -DHELLO

bye_SOURCES = bye.c
bye_CPPFLAGS = -DBYE
```

The local setting overrides the global one, thus

```
hello_CPPFLAGS = $(AM_CPPFLAGS) -Dmyflags
```

may be a good idea.

It is even possible to compile the same sources with different flags:

```
noinst_PROGRAMS = hello bye

hello_SOURCES = generic-greeting.c
hello_CPPFLAGS = -DHELLO

bye_SOURCES = generic-greeting.c
bye_CPPFLAGS = -DBYE
```

Perhaps you're wondering why the above examples used `AM_CPPFLAGS` instead of the normal `CPPFLAGS`? The reason for this is that the variables `CFLAGS`, `CPPFLAGS`, `CXXFLAGS` etc. are considered *user variables* which may be set on the command line:

```
make CXXFLAGS="-O2000"
```

This would override any settings in Makefile.am which might be necessary to build. Thus, if the variables should be set even if the user wishes to modify the values, you should use the `AM_*` version.

The real compile-command always uses both `AM_`*VAR* and *VAR* (or `progname_`*VAR* and *VAR*). Options that autoconf finds are stored in the user variables (so that they may be overridden).

Besides the three types of variables mentioned so far (user-, automake- and program-variables) there exists a fourth type by convention: variables of dependent libraries. These variables have the form *LIBRARY*`_`*VAR* and contain flags necessary to build programs or libraries which depend on that library. They are usually included in *program*`_`*VAR*, like this:

```
foo_CPPFLAGS = $(AM_CPPFLAGS) $(SUPERLU_CPPFLAGS)
```

If all programs build by the same makefile depend on a library, *program*`_`*VAR* can be included in `AM_`*VAR* instead:

```
AM_CPPFLAGS = @AM_CPPFLAGS@ $(SUPERLU_CPPFLAGS)
```

There are five classes of variables in automake-generated makefiles:

**automake** Example: `AM_CPPFLAGS`. These variables are usually undefined by default and the developer may assign them default values in the `Makefile.am`:

```
AM_CPPFLAGS = -DMY_DIR=‘pwd‘
```

**Automake** variables are not automatically substituted by `configure`, though it is common for the developer to `AC_SUBST` them. In this case a different technique must be used to assign values to them, or the substituted value will be ignored. See the **configure-substituted** class below. The names of **automake** variables begin with `AM_` most of the time, but there are some variables which don't have that prefix. These variables give defaults for **target-specific** variables.

**configure-substituted** Example: `srcdir`. Anything can be made a **configure-substituted** variable by calling `AC_SUBST` in `configure.ac`. Some variables always substituted by autoconf[1] or automake, others are only substituted when certain autoconf macros are used. In Dune, it is quiet common to substitute **automake** variables:

```
AC_SUBST(AM_CPPFLAGS, $DUNE_CPPFLAGS)
```

[1] autoconf manual, section "Preset Output Variables"

The value substituted by `configure` can be augmented in the `Makefile.am` like this:

```
AM_CPPFLAGS = @AM_CPPFLAGS@ -DMY_DIR=‘pwd‘
```

**target-specific** Example: *target*`_CPPFLAGS`. The names of these variables are of the form canonical target name followed by an underscore followed some uppercase letters. If there is a **automake** variable corresponding to this **target-specific** variable, the uppercase letters at the end of the name usually correspond to the name of that **automake** variable. These variables provide target-specific information. They are defined by the developer in the `Makefile.am` and are documented in the automake manual. If there is corresponding a **automake** variable it provides a default which is used when the **target-specific** variable is not defined. Example definition:

```
false_SOURCES = true.c
false_CPPFLAGS = $(AM_CPPFLAGS) -DEXIT_CODE=1
```

This example also shows how to include the value of the corresponding **automake** variable.

**user** Example: `CPPFLAGS`. These variables are for the user to set on the make command line:

```
make CPPFLAGS=-DNDEBUG
```

They usually augment some **target-specific** or **makefile-default** variable in the build rules. Often these variables are *precious*[2], and the user can tell `configure` what values these variables should have. These variables are **configure-substituted**.

The developer should never set this variables in the `Makefile.am`, because that would override the user-provided values given to configure. Instead, `configure.ac` must be tweaked to set a different default if the user does not give a value to `configure`.

**external-library** Example: *LIB*`CPPFLAGS`. These variables contain settings needed when using external libraries in a target. They should be included in the value for the corresponding **target-specific** variable

```
testprog_CPPFLAGS = $(AM_CPPFLAGS) $(SUPERLUCPPFLAGS)
```

or the **makefile-default** variable

```
AM_CPPFLAGS = @AM_CPPFLAGS@ $(SUPERLUCPPFLAGS)
```

Values for these variables are determined by `configure`, thus they are **configure-substituted**. Usually, `configure.ac` must call the right autoconf macro to determine these variables.

Note that the variable name with an underscore *LIB*`_CPPFLAGS` is not recommended[3], although this pattern is common.

Commonly used variables are:

**preprocessor flags** These flags are passed in any build rule that calls the preprocessor. If there is a **target-specific** variable *target*`_CPPFLAGS` defined, the flags are given by

```
$(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(target_CPPFLAGS) $(CPPFLAGS)
```

[2] autoconf manual, `AC_ARG_VAR`

[3] Autoconf manual, section “Flag Variables Ordering”

otherwise

```
$(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
```

is used.

`DEFS` Class: **configure-substituted**. Contains all the preprocessor defines from `AC_DEFINE` and friends. If a `config.h` header is used, contains just the value `-DHAVE_CONFIG_H` instead.

`DEFAULT_INCLUDES` Class: **configure-substituted**. This variables contains a set of default include paths: `-I.`, `-I$(srcdir)`, and an path to the directory of `config.h`, if that is used.

`INCLUDES` Class: **automake**. This is an obsolete alternative to `AM_CPPFLAGS`. Use that instead.

*target*`_CPPFLAGS` Class: **target-specific**. Target-specific preprocessor flags. If this variable exists, it overrides `AM_CPPFLAGS` and causes the renaming of object files[4].

`AM_CPPFLAGS` Class: **automake**. This is the makefile default for any preprocessor flags.

`CPPFLAGS` Class: **user**, **configure-substituted**. Flags given by the user, either to `configure` or when invoking `make`. If the user didn't provide any value to `configure`, it may contain debugging and optimization options per default (like `-DNDEBUG`). The value of `CPPFLAGS` always appears after the other preprocessor flags.

*LIB*`CPPFLAGS` Class: **external-library**. Preprocessor flags when building with library *LIB*. This variable should be include in *target*`_CPPFLAGS` or `AM_CPPFLAGS` in the `Makefile.am`.

**C-compiler flags** These flags are passed in any build rule that calls the C compiler or the C linker. If there is a **target-specific** variable *target*`_CFLAGS` defined, the flags are given by

```
$(target_CFLAGS) $(CFLAGS)
```

otherwise

```
$(AM_CFLAGS) $(CFLAGS)
```

is used.

*target*`_CFLAGS` Class: **target-specific**. Target-specific C compiler flags. If this variable exists, it overrides `AM_CFLAGS` and causes the renaming of object files[5].

`AM_CFLAGS` Class: **automake**. This is the makefile default for any C compiler flags.

`CFLAGS` Class: **user**, **configure-substituted**. Flags given by the user, either to `configure` or when invoking `make`. If the user didn't provide any value to `configure`, it may contain debugging, optimization and warning options per default (like `-g -O2 -Wall`). The value of `CFLAGS` always appears after the other C compiler flags.

**C++-compiler flags** These flags are passed in any build rule that calls the C++ compiler or the C++ linker. If there is a **target-specific** variable *target*`_CXXFLAGS` defined, the flags are given by

```
$(target_CXXFLAGS) $(CXXFLAGS)
```

otherwise

[4] automake manual, "Why are object files sometimes renamed?"
[5] automake manual, "Why are object files sometimes renamed?"

```
$(AM_CXXFLAGS) $(CXXFLAGS)
```

is used.

*target*_CXXFLAGS Class: **target-specific**. Target-specific C++ compiler flags. If this variable exists, it overrides AM_CXXFLAGS and causes the renaming of object files[6].

AM_CXXFLAGS Class: **automake**. This is the makefile default for any C++ compiler flags.

CXXFLAGS Class: **user**, **configure-substituted**. Flags given by the user, either to `configure` or when invoking `make`. If the user didn't provide any value to `configure`, it may contain debugging, optimization and warning options per default (like `-g -O2 -Wall`). The value of CXXFLAGS always appears after the other C++ compiler flags.

**linker flags** These flags are passed in any build rule that calls the linker. If there is a **target-specific** variable *target*_LDFLAGS defined, the flags are given by

```
$(target_LDFLAGS) $(LDFLAGS)
```

otherwise

```
$(AM_LDFLAGS) $(LDFLAGS)
```

is used. These variables are inappropriate to pass any options or parameters that specify libraries of object files, in particular `-L` or `-l` or the libtool options `-dlopen` and `-dlpreopen`. Use a variable from the *libraries to link to* set to do that.

*target*_LDFLAGS Class: **target-specific**. Target-specific C++ compiler flags. If this variable exists, it overrides AM_LDFLAGS. The existence of this variable does *not* cause renaming of object files[7].

AM_LDFLAGS Class: **automake**. This is the makefile default for any linker flags.

LDFLAGS Class: **user**, **configure-substituted**. Flags given by the user, either to `configure` or when invoking `make`. If the user didn't provide any value to `configure`, it may contain debugging, optimization and warning options per default. The value of LDFLAGS always appears after the other linker flags.

*LIB*LDFLAGS Class: **external-library**. Linker flags needed when linking to library *LIB*. This variable should be include in *target*_LDFLAGS or AM_LDFLAGS in the `Makefile.am`.

**libraries to link to** These variables are used to determine the libraries and object files to link to. They are passed whenever the linker is called. When linking a program, extra libraries and objects to link to are given by

```
$(target_LDADD) $(LIBS)
```

If the **target-specific** variable *target*_LDADD is not defined, automake supplies

```
target_LDADD = $(LDADD)
```

When linking a library, extra libraries and objects to link to are given by

[6] automake manual, "Why are object files sometimes renamed?"
[7] automake manual, "Why are object files sometimes renamed?"

```
$(target_LIBADD) $(LIBS)
```

If the **target-specific** variable *target*_LIBADD is not defined, automake defines it empty

```
target_LIBADD =
```

Libraries and objects to link to must be given in reverse order: a library or object file must come before the libraries or object files it depends on on the linker command line. Thus the value of the LIBS variable is included after the value of the *target*_LDADD or *target*_LIBADD variable.

In general, any linker flags and argument that specify libraries and object files should be included in these variables, and nothing else. In particular that means library and object file names, the options -L and -l, and the libtool options -dlopen and -dlpreopen. The option -L should come directly before any -l options it sets the linker path for, otherwise a path set by another -L option may take precedence, which may happen to contain a library by the same name.

*target*_LDADD Class: **target-specific**. Target-specific libraries and objects to link to *for programs*. If this variable does not exist, it defaults to $(LDADD).

LDADD Class: **automake**. Libraries and objects to link to *for programs*. Default for *target*_LDADD.

*target*_LIBADD Class: **target-specific**. Target-specific objects to link to *for libraries*. If the target is a libtool library, then other libtool libraries may also be specified here. This variable has no makefile-wide default, if it does not exist the empty value is assumed.

LIBS Class: **automake**, **configure-substituted**. Libraries discovered by configure.

*LIB*LIBS Class: **external-library**. Libraries and object files needed to linking against library *LIB*, including that library itself. This variable should be include in *target*_LDADD, LDADD, or *target*_LIBADD in the Makefile.am.

### Individual library variables

**MPI** The DUNE_MPI macro sets the following variables with the help of the macros ACX_MPI and MPI_CONFIG: For compilation with the MPI compiler MPICC and MPILIBS. These are not used in DUNE except that MPICC may be set on the configure command line to select which MPI installation to use. For compilation with the standard compiler it sets DUNEMPICPPFLAGS, DUNEMPILDFLAGS and DUNEMPILIBS, and the deprecated variables MPI_CPPFLAGS and MPI_LDFLAGS (note there is no MPI_LIBS). Unfortunately with most MPI implementations it is impossible to obtain the linker flags separately from the libraries to link to. Therefore, this macro stuffs everything into DUNEMPILIBS, which has the advantage that it works and the disadvantage that users are unable to overwrite the linker flags. If that is a problem users should set these variables themselves on the configure command line.

In addition, this macro substitutes MPI_VERSION a text string identifying the detected version of MPI. It defines the following preprocessor defines: MPI_2, defined if the detected MPI supports the MPI-2 standard. HAVE_MPI, 1 if MPI is detected and enabled. It also defines the automake conditional MPI.

DUNE **modules** For each DUNE module there are the variables *MODULE*_CPPFLAGS, *MODULE*_LDFLAGS and *MODULE*_LIBS. They contain everything to use that module with its most basic functionality. For instance, for dune-grid they do not contain the stuff for MPI, Alberta, ALU Grid or UG, even

if those were detected. They do contain the stuff for `dune-common`, possibly with duplicates removed, since that is absolutely required for the operation of `dune-grid`. Example use:

```
foo_SOURCES = foo.cc
foo_CPPFLAGS = $(AM_CPPFLAGS) \
    $(UG_CPPFLAGS) \
    $(DUNE_GRID_CPPFLAGS)
foo_LDFLAGS = $(AM_LDFLAGS) \
    $(UG_LDFLAGS) \
    $(DUNE_GRID_LDFLAGS)
foo_LDADD = \
    $(DUNE_GRID_LIBS) \
    $(UG_LIBS) \
    $(LDADD)
```

Note that there are no such variables for the current module – these variables are used in the process of building the current module, so that module is incomplete when detecting these variables. Note also that by "`DUNE` module" we mean a software package which uses the `DUNE` build system, not one of the official dune modules.

**Basic** `DUNE` To use the basic functionality of all detected `DUNE` modules, the variables `DUNE_CPPFLAGS`, `DUNE_LDFLAGS` and `DUNE_LIBS` may be used. They collect the contents of all `DUNE` module variables, possibly with duplicates removed.

**Extended** `DUNE` To use `DUNE` with all functionality that requires external libraries, the variables `ALL_PKG_CPPFLAGS`, `ALL_PKG_LDFLAGS` and `ALL_PKG_LIBS` may be used. They provide everything necessary to build with any external library detected by configure. In the case of Alberta a choice must be made between 2D and 3D. Here the `ALL_PKG_*` variables just follow the choice of the corresponding `ALBERTA_*` variables.

### Conditional builds

Some parts of `DUNE` only make sense if certain add-on packages were found. autoconf therefore defines *conditionals* which automake can use:

```
if OPENGL
  PROGS = hello glhello
else
  PROGS = hello
endif

hello_SOURCES = hello.c

glhello_SOURCES = glhello.c hello.c
```

This will only build the *glhello* program if OpenGL was found. An important feature of these conditionals is that they work with any make program, even those without a native *if* construct like GNU-make.

### Default targets

An automake-generated Makefile does not only know the usual *all*, *clean* and *install* targets but also

**tags** travel recursively through the directories and create TAGS-files which can be used in many editors to quickly find where symbols/functions are defined (use emacs-format)

**ctags** the same as "tags" but uses the vi-format for the tags-files

**dist** create a distribution tarball

**check** run a set of regression tests

**distcheck** create a tarball and do a test-build if it really works

### 5.1.2 Building Documentation

If you want to build documentation you might need additional make rules. `DUNE` offers a set of predefined rules to create certain kinds of documentation. Therefor you have to include the appropriate rules from the `am/` directory. These rules are stored in the `dune-common/am/` directory. If you want to use these any of these rules in your `DUNE` module or application you will have to create a symbolic link to `dune-common/am/`. The creation of this link should be done by the `dune-autogen` script.

The build system automatically gives you two targets related to the documentation:

**doc** make the documentation,

**doc-clean** clean up all documentation-related stuff.

#### doxygen

The source code documentation system `doxygen` is the preferable way to document your source and header files.

In order to build `doxygen` documentation you can include `$(top_srcdir)/am/doxygen`. Additionally you have create a file `Doxylocal` which contains your local `doxygen` configuration.

Your `doxygen` documentation should be located in the subdirectory `doc/doxygen/` (see "Coding Style" in the section "Developing Dune" on `http://www.dune-project.org/` for details). *After running `duneproject` the basic setup is already done.*

You should only have one `doxygen` directory and the files are automatically installed into `$prefix/share/doc/$modulename/doxygen/`. If for any reason you really have to change the installation path you can set the variable `doxygendir` *after* including `am/doxygen`.

The file `doc/doxygen/Doxylocal` contains the basic information where header and source files are located in your project. Usually you will not have to adjust this file, it is already created by `duneproject`. It only contains the very basic information. During the `dune-autogen` run the script `dunedoxynize` uses the information contained in `Doxylocal`, merges them with the global `DUNE doxygen` styles and writes `Doxyfile.in`, which will be translated into a full `Doxyfile` during the `configure` run. For details about the configuration of `doxygen` and about documenting your source code we refer to the `doxygen` web-site `http://www.doxygen.org/`.

#### html pages

Webpages are created from wml sources, using the program `wml` (`http://thewml.org/`).
`$(top_srcdir)/am/webstuff` contains the necessary rules.

Add all `html` files to the `PAGES` variable to build and install them.

**Listing 1 (File Makefile.am)**

```
# $Id: Makefile.am 6411 2011-03-29 21:07:41Z sander $

# also build these sub directories
SUBDIRS = doxygen buildsystem

# setting like in dune-web
CURDIR=doc
# position of the web base directory,
# relative to $(CURDIR)
BASEDIR=..
EXTRAINSTALL=example.opts

# install the html pages
docdir=$(datadir)/doc/dune-common
DOCFILES = $(PAGES)
DOCFILES_EXTRA = example.opts

EXTRA_DIST = $(PAGES) example.opts

# include rules for wml -> html transformation
include $(top_srcdir)/am/webstuff

# include further rules needed by Dune
include $(top_srcdir)/am/global-rules
```

#### LaTeXdocuments

In order to compile LaTeXdocuments you can include `$(top_srcdir)/am/latex`. This way you get rules for creation of DVI files, PS files and PDF files.

#### SVG graphics

SVG graphics can be converted to png, in order to include them into the web page. This conversion can be done using inkscape (`http://www.inkscape.org/`). `$(top_srcdir)/am/inkscape.am` offers the necessary rules.

### 5.1.3 Automatic testing

Dune offers several special `make` targets, which help you find problems in your build system configuration, or in your code.

**check** You can define lists of regression tests in your `Makefile.am`. These are run when you call `make check`.

**distcheck** This target is already defined by automake. It creates a tarball, unpacks it, tries to do an out-of-source build and runs the regression tests against this build.

**sourcescheck** This target tries to make sure that you don't forget to install any important headers or source files.

**headercheck** This target tries to make sure that your header files can be parsed and are self-contained.

#### The check target

TODO...

#### The sourcescheck target

TODO...

#### The headercheck target

TODO...

### 5.2 configure.ac

`configure.ac` is a normal text file that contains several `autoconf` macros. These macros are evaluated by the `m4` macro processor and transformed into a shell script.

**Listing 2 (File dune-common/configure.ac)**

```
#! /bin/bash
# $Id: configure.ac 6339 2011-02-09 22:00:22Z christi $
# Process this file with autoconf to produce a configure script.

DUNE_AC_INIT # gets module version from dune.module file
AM_INIT_AUTOMAKE
AC_CONFIG_SRCDIR([dune/common/stdstreams.cc])
AM_CONFIG_HEADER([config.h])

# add configure flags needed to create log files for dune-autobuild
DUNE_AUTOBUILD_FLAGS
# check all dune dependecies and prerequisits
DUNE_CHECK_ALL

# preset variable to path such that #include <dune/...> works
AC_SUBST([DUNE_COMMON_ROOT], '$(abs_top_srcdir)')
AC_SUBST([DUNE_COMMON_BIN], '$(abs_top_srcdir)/bin/')
AC_SUBST([AM_CPPFLAGS], '-I$(top_srcdir)')
AC_SUBST([LDADD], '$(top_builddir)/lib/libdunecommon.la')

# write output
AC_CONFIG_FILES([Makefile
     lib/Makefile
     bin/Makefile
     dune/Makefile
     dune/common/Makefile
     dune/common/test/Makefile
     dune/common/exprtmpl/Makefile
     dune/common/parallel/Makefile
     dune/common/parallel/test/Makefile
     doc/Makefile
     doc/doxygen/Makefile
     doc/doxygen/Doxyfile
     doc/buildsystem/Makefile
     m4/Makefile
     am/Makefile
     bin/check-log-store
     dune-common.pc])
AC_OUTPUT

# make scripts executable
chmod +x bin/check-log-store

# print results
DUNE_SUMMARY_ALL
```

We offer a set of macros that can be used in your `configure.ac`:

- DUNE_CHECK_ALL runs all checks usually needed by a *DUNE module*. It checks for all dependencies and suggestions and for their prerequisites. In order to make the dependencies known to configure dune-autogen calls dunecontrol m4create and write a file dependencies.m4.
- DUNE_SYMLINK creates symlink $(top_srcdir)/dune → $(top_srcdir). The programming guidelines (3) require that the include statements be like #include <dune/...>. If your module has a directory structure $(top_srcdir)/foo, you will need such a link. However, you are encouraged to store the files directly in a directory structure $(top_srcdir)/dune/foo in order to avoid any inconvenience when copying the files. This will also eliminate the necessity for DUNE_SYMLINK.
- DUNE_AUTOBUILD_FLAGS adds configure flags needed to create log files for dune-autobuild. If you want to add your module to the dune-autobuild system, you have to call this macro.
- DUNE_SUMMARY_ALL prints information on the results of all major checks run by DUNE_CHECK_ALL.

DUNE_CHECK_ALL defines certain variables that can be used in the configure script or in the Makefile.am:

- DUNE *MODULE* _CPPFLAGS
- DUNE *MODULE* _LDFLAGS
- DUNE *MODULE* _LIBS
- DUNE *MODULE* ROOT

The last step to a complete configure.ac is that you tell autoconf which files should be generated by configure. Therefore you add an AC_CONFIG_FILES([*WhiteSpaceSeparatedListOfFiles*]) statement to your configure.ac. The list of files should be the list of files that are to be generated, not the input—i.e. you would write

```
AC_CONFIG_FILES([Makefile doc/Makefile])
```

instead of

```
AC_CONFIG_FILES([Makefile.in doc/Makefile.in])
```

After you told autoconf which files to create you have to actually trigger their creation with command AC_OUTPUT.

## 5.3 Using configuration information provided by configure

The ./configure script in the module produces a file config.h that contains information about the configuration parameters, for example which of the optional grid implementations is available and which dimension has been selected (if applicable). This information can then be used at compile-time to include header files or code that depend on optional packages.

As an example, the macro HAVE_UG can be used to compile UG-specific code as in

```
#if HAVE_UG
#include"dune/grid/uggrid.hh"
#endif
```

It is important that the file config.h is the first include file in your application!

## 5.4 dune-autogen

The `dune-autogen` script is used to bring the freshly checked out module into that state that you expect from a module received via the tarball. That means it runs all necessary steps so that you can call `configure` to setup your module. In the case of `DUNE` this means that `dune-autogen` runs

- `libtoolize` (prepare the module for `libtool`)
- `dunecontrol m4create` (create an m4 file containing the dependencies of this module)
- `aclocal` (collect all `autoconf` macros needed for this module)
- `autoheader` (create the `config.h.in`)
- `automake` (create the `Makefile.in`)
- `autoconf` (create `configure`)

If needed it will also create the symbolic link to the `dune-common/am/` directory (see 5.1.2).

## 5.5 m4 files

`m4/` files contain macros which are then composed into `configure` and are run during execution of `configure`.

### private m4 macros

You can add new tests to configure by providing additional macro files in the directory `module/m4/`.

### dependencies.m4

`$(top_srcdir)/dependencies.m4` hold all information about the dependencies and suggestions of this module. It is an automatically generated file. It is generated by `dunecontrol m4create`.

### m4 module checks

For each dependencies of your module *MODULE*`_CHECKS` and *MODULE*`_CHECK_MODULE` is called. Last *MODULE*`_CHECKS` is called for your module, in order to check all prerequisites for your module.

What you just read implies that you have to provide the two macros *MODULE*`_CHECKS` and *MODULE*`_CHECK_MODULE` for your module. These should be written to a `m4/*.m4` file.

Here follows an example for the module `dune-foo`:

```
dnl -*- autoconf -*-
# Macros needed to find dune-foo and dependent libraries. They are
# called by the macros in ${top_src_dir}/dependencies.m4, which is
# generated by "dunecontrol autogen"

# Additional checks needed to build dune-foo
# This macro should be invoked by every module which depends on
# dune-foo, as well as by dune-foo itself
AC_DEFUN([DUNE_FOO_CHECKS])

# Additional checks needed to find dune-foo
# This macro should be invoked by every module which depends on dumux, but
# _not_ by dune-foo itself
AC_DEFUN([DUNE_FOO_CHECK_MODULE],[
  DUNE_CHECK_MODULES([dune-foo],        dnl module name
                     [foo/foo.hh],      dnl header file
```

```
                    [Dune::FooFnkt])   dnl symbol in libdunefoo
])
```

The first one calls all checks required to make use of `dune-foo`. The dependency checks are not to be included, they are run automatically. The second macro tells how to check for your module. In case you are only writing an application and don't want to make this module available to other modules, you can just leave it empty. If you have to provide some way to find your module. The easiest is to use the `DUNE_CHECK_MODULES` macro, which is defined in `dune-common/m4/dune.m4`.

## 6 Building Sets of Modules Using `dunecontrol`

`dunecontrol` helps you building the different `DUNE` modules in the appropriate order. Each module has a `dune.module` file which contains information on the module needed by `dunecontrol`.

`dunecontrol` searches for `dune.module` files recursively from where you are executing the program. For each `DUNE` module found it will execute a `dunecontrol` command. All commands offered by `dunecontrol` have a default implementation. This default implementation can be overwritten and extended in the `dune.module` file.

The commands you are interested in right now are

- `autogen` runs `dune-autogen` for each module. A list of directories containing `dune.module` files and the parameters given on the command line are passed as parameters to `dune-autogen`.
- `configure` runs `configure` for each module. `--with-dune`*`module`* parameters are created for a set of known `DUNE` modules.
- `make` runs `make` for each module.
- `all` runs `dune-autogen`, `configure` and `make` for each module.

In order to build `DUNE` the first time you will need the `all` command. In pseudo code `all` does the following:

```
foreach ($module in $Modules) {
  foreach (command in {autogen,configure,make) {
    run $command in $module
  }
}
```

This differs from calling

```
dunecontrol autogen
dunecontrol configure
dunecontrol make
```

as it ensures that i.e. `dune-common` is fully built before `configure` is executed in `dune-grid`. Otherwise `configure` in `dune-grid` would complain that `libcommon.la` from `dune-common` is missing.

Further more you can add parameters to the commands; these parameters get passed on to the program being executed. Assuming you want to call `make clean` in all `DUNE` modules you can execute

```
dunecontrol make clean
```

### opts files

You can also let `dunecontrol` read the command parameters from a file. For each command you can specify parameters. The parameters are stored in a variable called *`COMMAND`*`_FLAGS` with *`COMMAND`* written in capital letters.

**Listing 3 (File example.opts)**

```
# use these options for configure if no options a provided on the cmdline
AUTOGEN_FLAGS="--ac=2.50␣--am=-1.8"
CONFIGURE_FLAGS="CXX=g++-3.4␣--prefix='/tmp/Hu␣Hu'"
MAKE_FLAGS=install
```

When you specify an opts file and command line parameters

```
dunecontrol --opts=some.opts configure --with-foo=bar
```

dunecontrol will ignore the parameters specified in the opts file and you will get a warning.

#### environment variables

You can further control the behavior of dunecontrol by certain environment variables.

- DUNE_CONTROL_PATH specifies the paths, where dunecontrol is searching for modules. All entries have to be colon separated and should point to either a directory (which is search recursively for dune.module files) or a directly dune.module file.
- DUNE_OPTS_FILE specifies the opts file that should be read by dunecontrol. This variable will be overwritten by the --opts= option.
- MAKE tells dunecontrol which command to invoke for 'make'. This can be useful for example, if you want to use *gmake* as a make drop-in.
- GREP tells dunecontrol which command to invoke for 'grep'.

#### dune.module

The dune.module file is split into two parts. First we have the parameter section where you specify parameters describing the module. Then we have the command section where you can overload the default implementation of a command called via dunecontrol.

**Listing 4 (File dune.module)**

```
# parameters for dune control
Module: dune_grid
Depends: dune_common
Suggests: UG Alberta Alu3d

# overload the run_configure command
run_configure () {
  # lets extend the parameter list $CMD_FLAGS
  if test "x$HAVE_UG" == "xyes"; then
    CMD_FLAGS="$CMD_FLAGS␣\"--with-ug=$PATH_UG\""
  fi
  if test "x$HAVE_Alberta" == "xyes"; then
    CMD_FLAGS="$CMD_FLAGS␣\"--with-alberta=$PATH_Alberta\""
  fi
  if test "x$HAVE_Alu3d" == "xyes"; then
    CMD_FLAGS="$CMD_FLAGS␣\"--with-alugrid=$PATH_Alu3d\""
  fi
  # call the default implementation
  run_default_configure
}
```

The parameter section will be parsed by `dunecontrol` will effect i.e. the order in which the modules are built. The parameters and their values are separated by colon. Possible parameters are

- `Module` (*required*) is the name of the module. The name is of the form `[a-zA-Z0-9_]+`.
- `Depends` (*required*) takes a space separated list of required modules. This module is not functional without these other modules.
- `Suggests` (*optional*) takes a space separated list of optional modules. This module is functional without these other modules, but can offer further functionality if one or more of the suggested modules are found.

The command section lets you overload the default implementation provided by `dunecontrol`. For each command `dunecontrol` call the function `run_`*`command`*. The parameters from the command line or the opts file are store in the variable `$CMD_FLAGS`. If you just want to create additional parameters you can add these to `$CMD_FLAGS` and then call the default implementation of the command via `run_default_`*`command`*.

## 7 Further documentation

**automake & Makefile.am**
`http://www.gnu.org/software/automake/manual/`
The `automake` manual describes in detail how to write and maintain a `Makefile.am` and the usage of `automake`.

**autoconf & configure.ac**
`http://www.gnu.org/software/autoconf/manual/`
The `autoconf` manual covers the usage of `autoconf` and how to write `configure`.ac files (sometimes they are called `configure.in`).

**Autoconf Macro Archive**
`http://autoconf-archive.cryp.to/`
The Autoconf Macro Archive provides macros that can be integrated in your `configure.ac` in order to search for certain software. These macros are useful to many software writers using the autoconf tool, but too specific to be included into autoconf itself.

**doxygen**
`http://www.doxygen.org/`
The `doxygen` website offers documentation on how to document your source code and also on the configuration parameters in your `Doxylocal` file.

**libtool**
`http://www.gnu.org/software/libtool/manual.html`
The `libtool` manual offers further information on the usage of `libtool` package and gives a good overview of the different problems/aspects of creating portable libraries.

**autobook**
`http://sources.redhat.com/autobook/`
The autobook is a complete book describing the GNU toolchain (`autoconf`, `automake` and `libtool`). It contains many recipes on how to use the autotools. The book is available as an online version.

**dune-project**
`http://www.dune-project.org/`
The official homepage of `DUNE`.