Bakefile version 0.2.11

Custom manual made by Michael Gautier 9/23/2019

Made with pandoc.

Converted from html to odt then exported to pdf from LibreOffice.

Hello, world

By far the simplest way to generate makefiles using Bakefile is to use so-called presets which are prepared skeletons of bakefiles that get you started quickly. Let's see how it works on an example of the famous *Hello, world* program written in C and implemented in the `hello.c` file:

```
#include <stdio.h>

int main()
{
  printf("Hello, world!\n");
  return 0;
}
```

The bakefile needed to compile it,`hello.bkl`, looks like this:

```
<?xml version="1.0"?>
<makefile>

  <include file="presets/simple.bkl"/>

  <exe id="hello" template="simple">
    <sources>hello.c</sources>
  </exe>

</makefile>
```

Presets are included using the include directive. The structure of the file name is always the same: `presets/NAME-OF-PRESET.bkl`. In general, you can combine several presets, but in practice you must be careful when doing so. It's always a good idea to read the code for the preset before using it. The "simple" preset we include here defines a `DEBUG` option and a template called `simple`. Generated makefiles will allow the user to build all targets that are based on this template as either debug or release build.

Let's generate some makefiles now. The **bakefile** command is used to do it. For example:

```
$ bakefile -f msvc hello.bkl
```

That's all. This will creates VC++ makefile `makefile.vc`. Of course, you can change the name of output file if you don't like the default:

```
$ bakefile -f msvc -o makefile hello.bkl
```

Bakefile will also generate the `Makefile.in` files used by Autoconf:

```
$ bakefile -f autoconf hello.bkl
```

These are templates for makefiles. Autoconf also requires a `configure.ac` script (previously, `configure.in`), but Bakefile will not generate this for you. This script checks for platform features necessary to build the program; see the autoconf manual for details.

When producing autoconf format output, Bakefile will also generate a file called `autoconf_inc.m4` which defines macros needed by the generated Makefile.in files. To use this, call the `AC_BAKEFILE` macro within your `configure.ac` script.

A minimal `configure.ac` script for our example program would look like this:

```
AC_PREREQ(2.53)
AC_INIT([hello], [1.0], [author@example.com])
dnl pass some unique file file to AC_CONFIG_SRCDIR
AC_CONFIG_SRCDIR([autoconf_inc.m4])
AC_CANONICAL_HOST
DEBUG=0
```

```
AC_BAKEFILE([m4_include(autoconf_inc.m4)])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Note the part that sets the `DEBUG` variable. Any options declared in your bakefile must be set to some default value before calling `AC_BAKEFILE`. The simple.bkl preset defines the `DEBUG` option, so we have to give it a default value here.

While the above code will work, there's a better way to handle the debug option:

```
AC_PREREQ(2.53)
AC_INIT([hello], [1.0], [author@example.com])
dnl pass some unique file file to AC_CONFIG_SRCDIR
AC_CONFIG_SRCDIR([autoconf_inc.m4])
AC_CANONICAL_HOST

AC_ARG_ENABLE(debug,
              [  --enable-debug          Enable debugging information],
              USE_DEBUG="$enableval", USE_DEBUG="no")

if test $USE_DEBUG = yes ; then
  DEBUG=1
  dnl Bakefile doesn't touch {C,CPP,CXX,LD}FLAGS in autoconf format, we
  dnl have to do it ourselves. This will work with many compilers
  dnl (but not all, proper configure script would check if the compiler
  dnl supports it):
  CFLAGS="$CFLAGS -g"
else
  DEBUG=0
fi

AC_BAKEFILE([m4_include(autoconf_inc.m4)])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

You are ready to generate Autoconf's `configure` script now:

```
$          bakefilize --copy && aclocal && autoconf
```

Table of Contents

Targets

Like in traditional makefiles. Target is a single buildable entity, e.g. executable, library or DLL. See Chapter 4, *Targets* for details.

Variables

You can set and use variables in Bakefile in a way very similar to other make programs. Variables may be either global or local to the target they are bound to. Note that variables are expanded by Bakefile and do *not* appear as variables in generated native makefiles (compare with options).

Variables are expanded by typing **$(var)** (for some variable `var`) in makefile text. Same syntax is used to expand options and conditional variables, too.

Templates

It is often the case that several targets in the makefile share the same properties: for example, they are installed into same directory, use same compiler settings or include headers from same directories. Templates are a mechanism designed to eliminate such duplication from makefiles. In Bakefile, you simply declare that a target *derives* from template (or templates) and it will inherit all its properties. See description of template command.

Options

It is desirable that generated makefiles are configurable to some degree (especially with the `autoconf` backend). Bakefile makes it possible to declare so-called *options* using option command. Each format backend has its own way of presenting options and some formats may fail to support them at all. Some formats (such as Visual C++ project files backend) have only limited support of options. Typically, options are translated into variables in native makefiles and can be modified by the user.

This is how setting an option may look with `autoconf`:

```
$ ./configure --enable-debug
```

Or with VC++ makefiles:

```
C:\> nmake DEBUG=1
```

Conditions

Condition is a boolean expression that is used to conditionally determine values of variables and also to conditionally disable or enable parts of the makefile. Conditions are commonly used to differ generated native makefiles based on output format and user settings of options.

There are two types of conditions:

Weak

    The condition may be any Python expression that only uses variables defined with the set command and Python helper functions from available modules. The expression must evaluate to either 0 (false) or 1 (true).
    If it evaluates to 0, then the statement that has the condition associated with it is not processed. If it evalues to 1, that the statement is processed as if it had no condition.

Strong

These conditions depend on the value of an option. Their syntax is very limited compared to weak conditions, because the decisition (condition evaluation) is postponed until make-time -- i.e. generated native makefile contains the condition in some form.
The condition may only take the form of simple test for equality:
`OPTION=="VALUE"`
Here, `OPTION` is the name of an already defined option *with enumerated possible values* and `VALUE` is one of the option's values.
If a strong conditions is used with the set command, a conditional variable is created.

Examples of valid conditions:

```
<set var="NUM1">1</set>
<set var="NUM2">10</set>
<set var="RESULT_11" cond="NUM1+NUM2==11">yes</set>

<option name="BUILD">
  <values>debug,release</values>
  <default-value>release</default-value>
</option>

<set var="USE_DEBUG" cond="BUILD=='debug'">1</set>
<set var="USE_DEBUG" cond="BUILD=='release'">0</set>
```

Examples of invalid conditions:

```
<option name="NUM1">
  <default-value>1</default-value>
</option>
</set>
<set var="NUM2">10</set>
<set var="RESULT_11" cond="NUM1+NUM2==1">yes</set>

<option name="BUILD">
  <values>R D</values>
</option>

<set var="USE_DEBUG" cond="BUILD=='debug'">1</set>
     <!-- not in the list of values -->

<option name="BUILD2"></option>

<set var="USE_DEBUG2" cond="BUILD2=='debug'">1</set>
     <!-- not option with enumeration -->

<option name="BUILD3">
  <values>release debug</values>
</option>

<set var="USE_DEBUG3" cond="BUILD!='debug'">1</set>
     <!-- not equality test -->
```

See also: the section called “Conditional Variables”, set

Conditional Variables

Conditional variables are variables whose value differs depending on a condition. They are created by using the conditional form of set command. Unlike options, they can't be directly modified by user of native makefile. Unlike variables, they are not evaluated by Bakefile during processing (the value depends on values of options).

Summary of differences between options, variables and conditional variables:

Type	Value	Set by user
variable	constant	no

Type	Value	Set by user
option	variable	yes
conditional variable	variable (derived from some option)	no

Modules

Modules extend Bakefile with additional abilities. For example, standard configuration of Bakefile can't build Python modules. You must explicitly load `python` module which will add, among other things, `python-module` rule. Functionality is divided into modules so that generated makefiles are not cluttered with unused options and to avoid unnecessary *configure* checks.

Presets

Presets are pieces of Bakefile code that can be included in user bakefiles. Their purpose is to provide support for e.g. libraries or tools or to provide commonly used code snippets in convenient form.

For example, Bakefile contains the `simple` preset that can be used to quickly create makefiles with support for both debug and release builds.

Paths

Regardless of the operating system where Bakefile is running, the convention respected by all Bakefile tags and variables is to use Unix-style paths, i.e. to use forward slash (`/`) as the path separator.

Table of Contents

Bakefile targets correspond to native makefile targets: they are compiled programs, libraries, or more complex actions such as "install" or "dist". Target syntax is similar to command syntax:

```
<TYPE id="NAME" [template="TEMPLATE,..."] [template_append="TEMPLATE,..."]
      [cond="CONDITION"] [category="CATEGORY"]>
  SPECIFICATION
</TYPE>
```

There are six standard target types: exe, dll, module, lib, phony and action. You can define a new "rule" (that is, a target type) based on one of the standard rules using the define-rule command.

Each target requires a unique `id`. This ID is usually present in the generated makefile, so you can type `make myprogram` to create the target with the ID `myprogram`. The target's ID also controls the name of the output file.

`template` is an optional comma-separated list of IDs of templates that the target is derived from. `template_append` is an optional comma-separated list of templates that are *appended* to the target specification. (`template` inserts the template before the specification).

If the `cond` attribute is given, the target is only compiled if the named condition was met. (Follow that link to see the rules governing whether Bakefile evaluates the condition, or the native build system (e.g. `make`) evaluates it).

In addition to the ID, every target can have variables attached to it. These variables are only effective for the target; contrast global variables, which affect all targets. They can be used to override a global variable: for example, the `DLLEXT` variable is `.dll` for Windows makefiles, but you can override the variable locally for the `sharpen_plugin` target to be `.plugin`.

Target is described using `SPECIFICATION`, which is a list of set commands and *tags*. Tags are rule-specific constructs, so they come in several forms: they can list source files for an executable, set include directories, or define compiler flags. Unlike `set`, they don't set any specific variable, but rather set various variables in a *generator-specific way*. Tag syntax is almost identical to the `set` function, but without a variable name:

```
<TAGNAME>VALUE</TAGNAME>
```

Common tags are described in the section below. Tags specific to particular modules are described in Chapter 10, *Modules*. A small example of using tags:

```
<exe id="myprogram">
  <!-- set target-specific variable: -->
  <set var="SOME_VAR">value</set>
  <!-- three tags: -->
  <sources>file1.c myprogram.c utils.c</sources>
  <include>./includes</include>
  <define>USE_UNICODE</define>
</exe>
```

Unless the documentation says otherwise, you can use the same tag repeatedly with the same target.

The optional `category` attribute can be given to classify the target. Possible classifications are `all` (reserved for the `all` target of makefiles and cannot be used in user Bakefiles), `normal` for targets declared in Bakefiles, and `automatic` for targets that are created as a side-effect of Bakefile's processing (e.g. object file targets). The targets are sorted in the generated makefile according to the category: the `all` target is first, followed by `normal` targets, and then `automatic` targets.

Pseudo targets

Some rules don't declare real targets but so-called *pseudo targets*. Pseudo targets are processed as standard targets, but they don't appear in the generated makefile, have no action associated with them, can't depend on any other target, and can't be a dependency of another target. They can only modify the behavior of other targets. An example of a pseudo target is data-files.

The advantage of pseudo targets is that the `id` attribute is not required. The disadvantage is that they can't be conditional.

Standard Target Types ("Rules")

Description of builtin rules and their tags follows. Additional rules and tags are defined by modules, see Chapter 10, *Modules*.

exe

Builds a program.

Tag	Description
app-type	Use this tag to specify whether the executable is console application (`console`) or windowed one (`gui`). These two kinds of applications are linked differently on Windows. `<exe id="foo">` `<app-type>gui</app-type>` `<sources>foo.c bar.c</sources>` `</exe>`
exename	Set name of the executable. By default, the name is same as `id`, but it is sometimes useful to use different name to identify the executable in makefiles (`id`) and for created program file (`exename`). Physical filename is deriver from `exename` and format-specific extension (e.g. `.exe` on Windows).
stack	Set the size of the stack on platforms where it is possible. The default stack size varies for different platforms and compilers. With too small stack you may get an error indicating stack has overflowed. (Currently used by Watcom format only.)

lib

Tag	Description
libname	Set name of the library. By default, the name is same as `id`, but it is sometimes useful to use different name to identify the library in makefiles (`id`) and for created library file (`libname`). This tag does *not* set physical filename -- that is derived from `libname` and other generator-specific variables. `<lib id="foo">` `<libname>foo$(COMPILER)_$(DBGFLAG)</libname>` `<sources>foo.c bar.c</sources>` `</lib>`

dll

Tag	Description
dllname	Similar to libname tag on dll, but it affects the name of shared library/DLL.
libname	Similar to dllname, but used for import library on

Tag	Description
	Windows and .so symlink on Unix.
version	Portable platform-independent shared library version. *Not yet implemented.*
so_version	The value consists of three numbers separated by dots. Library name plus the first component of the version together form *soname* (e.g. `libfoo.so.1` on Linux) which is used for runtime resolution of dependencies. Libraries with different *sonames* are binary incompatible and cannot be used interchangeable. On the other hand, the remaining two components contain information about minor versions that don't change backward compatibility and aren't used by runtime linker. Note that this version number is strictly for runtime linker's use and for maintaining binary compatiblity, it should *not* be version number of your library. This tag is only implemented in the `autoconf` format.
mac_version	This tag is used to specify shared library version for runtime linker on Darwin platforms. Similarly to so_version, it expects three numbers separated by dots, but their interpretation is different. The value is passed as-is to compilers `-current_version` argument and the first two components are passed to `--compatibility_version`. See See Apple's ld documentation for details. This tag is only implemented in the `autoconf` and `xcode2` formats.

module

Builds loadable module (aka plugin). Unlike dll, this one does not create import library on Windows.

Tag	Description
dllname	See dllname.

phony

This type of target does nothing. It is usually used as convenience target together with depends tag to easily build set of targets. Standard target `all` is an example of phony target.

action

This is most generic rule. It allows you to execute arbitrary sequence of commands when building the target and can therefore be used to extend Bakefile with custom build rules. Note that this rule is not platform- and compiler-independent as the rest of rules.

Tag	Description

command	Adds command to list of commands to be executed when building the target.
	`<action id="manual.html">`
	`  <command>docbook2html manual.xml manual.html</command>`
	`</action>`

Tag	Description
is-phony	Marks the target as *phony*. A phony target is target whose name is not really the name of file produced by it, as is the case with "standard" targets. Instead, it's just a symbolic name for some commands to be executed. In other words, this target should not be considered by `make` when creating the file dependency graph. For an example of why this is useful, consider a `tags` target that runs the Unix `ctags` program. That program creates a file called `tags`, so after saying `make tags` once, `make` will always say that the tags target is up to date. By marking the target as phony, you disable the file dependency checking for it, so `make tags` always runs the `ctags` program.

subproject

Declares a subproject. This is typically another makefile in a subdirectory and is independent of its parent project. Therefore you can't use any variables or refer to targets from the parent project in a subproject.

Tag	Description
dir	Directory containing subproject. Parent makefile calls makefile in this directory.
target	Optional argument which specifies what target to build in the subproject.
installable	Whether **make install** (if supported by the format) should descend into subproject's directory. The value can only be `yes` or `no` and must be a constant expression.

Here is an example of how to use the subproject target:

```
<subproject id="examples">
  <dir>examples</dir>
  <installable>no</installable>
</subproject>
```

Common tags

These are tags you can use with any of the above target types. These tags are always available: it is not neccessary to load any module to use them.

Tag	Description	Availability
depends	This tag is used to express target's dependency on other targets. All targets in the `depends` tag must be built prior to building this target. The value is whitespace-separated list of target IDs. It is an error to specify a nonexistent target here. Note that the library tag implies `depends`. `<exe id="app">` `<sources>app.c</sources>` `<depends>setup</depends>`	all rules

Tag	Description	Availability
	`</exe>`	
dependency-of	Mark the target as dependency of target specified in tag's value. The value must be ID of existing target. This tag is opposite of depends. Following two examples have identical effect: `<exe id="setup"></exe>` `<exe id="app">` `<depends>setup</depends>` `</exe>` `<exe id="app"></exe>` `<exe id="setup">` `<dependency-of>app</` `dependency-of>` `</exe>` Note that only one of these tags should be used, it is not necessary to specify both of them.	all
headers	Specify (C/C++) header files used by the target. `<exe id="app">` `  <headers>app.h</headers>` `  <headers>utils.h` `additionalheader.h</headers>` `  <sources>app.c</sources>` `</exe>`	exe, dll, module, lib
depends-on-file	Mark the target as depending on a given file. Use this when the dependency isn't one of the other declared targets. This can be useful for example when the dependency is generated by a script: `<action id="generated-` `header.h">` `      <command>./header-` `generator.pl</command>` `      <depends-on-file>header-` `generator.pl</depends-on-file>` `</action>` In this example, the header-generator.pl script creates generated-header.h. You would want the script to be re-run any time it changes, since the change probably would make it generate different contents for that header file. But since generated-header.pl is not itself generated by the makefile, you cannot use the depends tag here.	all
objects-depend	Same as depends, except the dependency is added to all *object files* used to build the target instead	exe, dll, module, lib

Tag	Description	Availability
	of to the target itself. This is useful e.g. in combination with precompiled headers (which must be generated before any source file is compiled) or when a commonly used header file is generated by the makefile.	
dirname	Set name of directory where the target will be created. `BUILDDIR` is used by default.	exe, dll, module, lib

sources	Specify source files used to build the target.
	`<exe id="app">`
	`  <sources>app.c</sources>`
	`  <sources>utils.c utils2.c</sources>`
	`</exe>`
		exe, dll, module, lib

include	Add directory where the compiler should look for headers. This corresponds to the `-I` switch used by many compilers. Calls res-include with the same value.
	Example:
	`<exe id="hello">`
	`  <sources>hello.c</sources>`
	`  <include>../include/foo</include>`
	`</exe>`
		exe, dll, module, lib

define	Define C preprocessor macro. The value may be empty, in which case no flag is added. Calls res-define with same value.	exe, dll, module, lib

sys-lib	Link against specified library installed in the system. Note that this is *not* meant for linking in libraries that were built by the same makefile; use library for that. This command links against a library installed in the system or provided by the compiler and corresponds to the `-l` switch of Unix compilers.
	`<exe id="png2bmp">`
	`  <sources>png2bmp.c</sources>`
	`  <sys-lib>png</sys-lib>`
	`  <sys-lib>z</sys-lib>`
	`</exe>`
	The library name may be empty. Only one library may be given as the tag's argument; the following usage is incorrect:
	`<exe id="png2bmp">`
		exe, dll, module

Tag	Description	Availability

```
  <sources>png2bmp.c</sources>
  <sys-lib>png z</sys-lib>
<!-- INCORRECT -->
</exe>
```

Note that the name of the library in this tag is *not* a file name and must not include paths. Use lib-path to add a directory to the library search path.

Tag	Description	Availability
lib-path	Add a directory to the search path used by the compiler to find system libraries. This corresponds to the `-L` switch of Unix compilers. Example:	exe, dll, module

```
<exe id="hello">
  <sources>hello.c</sources>
  <!-- note that hardcoding
library paths like this is a
bad
       idea, it's done here
only for the sake of
simplicity;
       in real bakefile, an
<option> would be used -->

<lib-path>/usr/lib/mysql</lib-
path>
  <sys-lib>mysqlclient</sys-
lib>
</exe>
```

On a typical Unix system, this asks the linker to link the `hello` program against `libmysqlclient.so` and to search for it in the directory `/usr/lib/mysql` in addition to any other directories the linker is configured to use.

Tag	Description	Availability
library	Link against library compiled by this makefile. The value passed to this tag must be name of existing target. Compare sys-lib.	exe, dll, module

```
<lib id="mylib">
  <sources>lib1.c
lib2.c</sources>
</lib>
<exe id="myapp">
  <sources>main.c</sources>
  <library>mylib</library>
  <sys-lib>X11</sys-lib>
  <sys-lib>GL</sys-lib>
</exe>
```

Tag	Description	Availability
optimize	Set compiler's optimization level. May be one of `off` (no optimization), `speed` (generate fastest code) or `size` (smallest	exe, dll, module, lib

Tag	Description	Availability

	code).
	`<set var="OPTIMIZE_FLAG">`
	`  <if cond="BUILD=='release'">speed</if>`
	`  <if cond="BUILD=='debug'">off</if>`
	`</set>`
	`<exe id="myapp">`
	`  <optimize>$(OPTIMIZE_FLAG)</optimize>`
	`  <sources>main.c</sources>`
	`  <sys-lib>GL</sys-lib>`
	`</exe>`	

debug-info	Enable or disable debugging information. Can be either `on` or `off`.	exe, dll, module, lib
debug-runtime-libs	Enable or disable linking against debug version of C runtime. Can be either `on` or `off` and *must* appear after debug-info. If not specified, then debug runtime libraries are used if and only if debug-info was set to `on`. Note that this tag has effect only with Visual C++; other compilers respect only debug-info.	exe, dll, module, lib
debug-info-edit-and-continue	Enable or disable additional debugging information to support Edit and Continue feature of Visual C++ compilers. Can be either `on` or `off` (default). Only supported by Visual C++ project files, does nothing in other formats.	exe, dll, module, lib
arch	Set target CPU architecture. Note that this is not portable and should be avoided if possible. Accepted values are `i[3456]86`.	exe, dll, module, lib
pic	Tells the compiler whether to generate position-independent code or not. The default is to use PIC for DLLs and *not* use it for executables and static libraries, but you may want to change it if your static library may be linked into a DLL. Accepted values are `on` and `off`.	exe, lib
threading	Use `multi` to enable and `single` to disable multithreading in the application. This affects what libraries are linked into executable on some platforms.	exe, dll, module, lib
warnings	Sets warnings level for C/C++	exe, dll, module, lib

Tag	Description	Availability
	compiler. Possible values are `no`, `default` and `max`.	
precomp-headers	Can be `on` or `off`, enables or disables use of precompiled headers with compilers that support them.	exe, dll, module, lib
precomp-headers-file	Use this tag to fine-tune where precompiled headers are stored. The compiler must support this and the value passed to `precomp-headers-file` can be modified by Bakefile, e.g. by apending `.pch` extension to it.	exe, dll, module, lib
precomp-headers-gen	For compilers that support it, specify which source file should be used to generate precompiled headers. `<exe id="myapp">` `  <sources>foo.cpp bar.cpp</sources>` `  <precomp-headers>on</precomp-headers>` `  <precomp-headers-gen>foo.cpp</precomp-headers-gen>` `  <precomp-headers-header>mypch.h</precomp-headers-header>` `</exe>`	exe, dll, module, lib
precomp-headers-header	For compilers that support it, specify which header file should be used as the precompiled header. Typically, this header must be the first header included and it must be included by all source files. Some compilers (GCC in particular) ignore this, because they use different PCH model. `<exe id="myapp">` `  <sources>foo.cpp bar.cpp</sources>` `  <precomp-headers>on</precomp-headers>` `  <precomp-headers-gen>foo.cpp</precomp-headers-gen>` `  <precomp-headers-header>mypch.h</precomp-headers-header>` `</exe>`	exe, dll, module, lib
cxx-rtti	Enable or disable RTTI when compiling C++ sources. Can be	exe, dll, module, lib

Tag	Description	Availability
	either `on` or `off`.	
cxx-exceptions	Enable or disable C++ exceptions handling. Can be either `on` or `off`.	exe, dll, module, lib
cflags cxxflags cppflags	Add *compiler dependent* compilation flags to compiler flags.	exe, dll, module, lib
ldflags	Add *linker dependent* flags for the linker.	exe, dll, module, lib
ldlibs	Same as ldflags, but adds the flags *after* all flags specified using `ldflags`. This is useful when resolving command line order problems that gcc is prone to.	exe, dll, module
win32-res	Sets win32 resource (.rc) file for executable or DLL. The tag does nothing on platforms other than Windows. Compilation of the resource respects define and include tags on the target, as well as their resource specific counterparts res-define and res-include.	exe, dll, module
mac-res	Sets Mac resource (.r) file for executable or DLL. The tag does nothing on platforms other than Mac. Compilation of the resource respects define and include tags on the target, as well as their resource specific counterparts res-define and res-include.	exe, dll, module
symbian-res	Sets Symbian resource file for executable or DLL. The tag does nothing on platforms other than Symbian. Compilation of the resource respects define and include tags on the target, as well as their resource specific counterparts res-define and res-include.	exe, dll, module
res-include	Similar to include, but applies only to resources (mac-res, win32-res) and not to C/C++ sources.	exe, dll, module
res-define	Similar to define, but applies only to resources (mac-res, win32-res) and not to C/C++ sources.	exe, dll, module
clean-files	Adds files to list of files that are cleaned when **make clean** is run -- i.e. files created while building the target.	all
install-to	If used, then the target is installed into directory specified as tag's	exe, dll, module, lib

Tag	Description	Availability
	value by **make install** (and uninstalled from there by the **make uninstall** target).	
install-headers-to	If used, then the headers (see headers tag) of the target are installed into the directory specified as tag's value by **make install** (and uninstalled from there by the **make uninstall** target).	exe, dll, module, lib
install-if	Install (see install-to the target conditionally. The value must be well-formed condition. `<option name="INSTALL_HELLO">` `<values>0,1</values>` `<default-value>1</default-value>` `</option>` `<exe id="hello">` `<sources>hello.c</sources>` `<install-to>$(BINDIR)</install-to>` `<install-if>INSTALL_HELLO=='1'</install-if>` `</exe>`	exe, dll, module, lib
postlink-command	Use this tag to specify command(s) that must be executed after the target is linked. This can be used to e.g. add resources or strip debugging information.	exe, dll, module, lib
uid	Defines target's unique ID number for formats that need it. *FIXME: currently not implemented in any format; document use of type=symbian1 etc. once it is used by something*	exe, dll, module
msvc-guid	(Visual C++ project formats only.) Sets project's GUID.	exe, lib, dll, module

Autoconf Note

Many configuration options listed above are not supported by the Autoconf format (e.g. optimize, debug-info or arch. This is because `configure` is used to find appropriate compiler flags.

Table of Contents

Commands are top-level makefile constructs. They have following form:

```
<COMMAND [PROPERTY="VALUE", ...]>
    CONTENT
</COMMAND>
```

Here, `CONTENT` is either a text value (as in e.g. set) or XML subtree.

Makefile Commands

set

Sets a variable. There are two forms of the command. The first one is for setting variables unconditionally:

```
<set var="NAME" [append="APP"] [prepend="PREP"] [overwrite="OVERWRITE"]
     [scope="SCOPE"] [make_var="MAKEVAR"] [hints="HINTS"]>
  VALUE
</set>
```

The other one resembles *switch* statement known from C and is used to set the variable to one of possible values depending on certain condition:

```
<set var="NAME" [append="APP"] [prepend="PREP"] [overwrite="OVERWRITE"]
     [scope="SCOPE"] [make_var="MAKEVAR"] [hints="HINTS"]>
  <if cond="COND">VALUE</if>
  [
  <if cond="COND">VALUE</if>
  ...
  ]
</set>
```

If the second from is used then the variable is set to value from the first `if` node whose condition is met, or to empty string if no condition is met. Note that conditions within one `set` command *must be mutually exclusive.*

The value is any text that may contain variable expansions.

If an option with same name exists, the variable takes precedence and the option is shadowed by it. This behaviour allows you to hardcode values for some ruleset's options in the makefile or to specify the value on command line when running Bakefile.

Parameters:

var

Name of the variable to assign the value. Any constant expression is allowed for this attribute, not only literals.

```
<set var="postfix">world</set>
<set var="prefix">hello</set>

<!-- the following <set> tag will create a "hello_world" variable: -->
<set var="$(prefix)_$(postfix)">Hello world</set>
<echo>$(hello_world)</echo>
```

Required parameter

append

If 1, the value is appended to previous value of the variable if it is already defined, with a space inserted between them. If the variable wasn't defined yet, the command behaves as if append=0. Following two `set` commands are equivalent:

```
<set var="FOO" append="1">something</set>
<set var="FOO">$(FOO) something</set>
```

Default value: 0

prepend

If 1, the value is prepended in front of previous value of the variable if it is already defined (otherwise the command behaves as if prepend=0). Following two `set` commands are equivalent:

```
<set var="FOO" prepend="1">something</set>
<set var="FOO">something $(FOO)</set>
```

Default value: 0

cond

If present, the variable is set only if the condition is met. If the condition evaluates to 0, the variable is not set, if it evaluates to 1, the variable is set. If condition's value can't be determined at the time of makefile processing, a conditional variable is created instead of ordinary variable. See the section called “Conditions” for more details.

```
<set var="FILES">
  <if cond="BUILD=='debug'">foo_dbg.c</if>
  <if cond="BUILD=='release'">foo.c</if>
</set>
```

The condition can also value special value `target`, which can only be used within target specification. In that case parent target's condition is used (or `1` if there's no condition set on the target). The condition can also be "`target                              and`*condexpr*" in which case target's condition (if any) is combined with *condexpr*.

The string with condition may itself be a constant expression, so you can write this:

```
<set var="IsRelease">=='release'</set>
<set var="CondDebug">BUILD=='debug'</set>
<set var="FILES">
  <if cond="BUILD$(IsRelease)">foo_dbg.c</if>
  <if cond="$(CondDebug)">foo.c</if>
</set>
```

overwrite

If set to 0 and variable with this name already exists, then it's value is not changed (the default is to change it).
Default value: 1

scope

Specify scope of variable being set. Possible values are `local` (current target if the command is applied on a target, same as `global` otherwise), `global` or a name of existing target (in which case the variable is set on that target).
Can't be used with conditional variables.
Default value: local

make_var

If set to `1`, then the variable is preserved in the makefile instead of being substituted by Bakefile. This happens only if the output format supports it (FORMAT_HAS_VARIABLES is set to `1`) and if variable's value is not empty string. This settings is useful together with frequently used variables with long values, it helps reduce size of generated makefiles.
Default value: 0

hints

Comma-separated list of hint keywords. These hints are optional and Bakefile can (but doesn't have to) use them to better format generated makefiles. So far only `files` hint is supported. It tells Bakefile that the variable holds list of files and if it is either make or conditional variable, it is formatted in such way that only one file per line is written to the output (and therefore adding or removing files does only cause small differences).

Example:

```
<set var="APP_VERSION">1.0.3</set>
<set var="TAR_NAME">app-$(APP_VERSION).tar.gz</set>
```

See also: unset

unset

Unsets variable previously set by set. Note that you can only unset a *variable*, not an option or conditional variable.

```
<unset var="NAME"/>
```

Parameters:

var

The meaning is same as in set's properties.

option

Adds an option to the makefile.

```
<option name="NAME" [never_empty="NEVER_EMPTY"] [category="CATEGORY"]>
  [<default-value [force="FORCE"]>DEFVALUE</default-value>]
  [<description>DESC</description>]
  [<values>VALUES</values>]
  [<values-description>VALUES_DESC</values-description>]
</option>
```

`NAME` is variable name under which the option is used in the makefile (using same syntax as when expanding variables). `NAME` is required, the rest of parameters is optional.

`DEFVALUE` is default value of the option, if appliable. It can be used by format backends that don't support options and it is used as default in those that do. Use it whenever possible. Note that for options with listed values (see the `VALUES` parameter), the default value must be one of the values listed unless `FORCE` is set to `1`.

`FORCE` can be `0` (the default) or `1` to indicate that Bakefile should not check that the default value is in the list of allowed values. This is useful when you want to use e.g. a shell command as the default value (`$(shell some-command)`) or an environment variable `$(MYENVVAR)`. It is your responsibility to ensure that the default value is a legal value if you use `force=1`.

`NEVER_EMPTY` may be set to `1` to tell Bakefile that it can treat the option as non-empty variable. This is useful only rarely in situations when Bakefile requires some non-empty value as tag's argument.

`CATEGORY` may be set to provide Bakefile additional information about the option. Certain operations (typically substitutions) may fail when applied to options unless all of its possible values are known. Because many tags use substitutions internally, this can be very limiting; the category hint can be used to work around most common problems. Possible values are `unspecified` (the default) and `path`, which indicates that the option will contain valid *native*, non-empty path name. An option with category set to `path` can be used as argument to tags like include.

`DESC` is human-readable description of the option, for use in comments.

`VALUES` is comma-separated list of all possible values the option can have. It is used by backends that don't support options (such as Visual C++ project files) to generate all possible configurations. It's use is highly recommended.

`VALUES_DESC` is comma-separated list of single-word description of corresponding values. It may be used only if `VALUES` were specified and both lists must have same length. These descriptions will show up in formats that don't support conditions, such as Visual C++ projects (the project will contain several configurations that will be described using these words).

template

Defines new template.

```
<template id="NAME" [template="TEMPLATE,..."]>
  SPECIFICATION
</template>
```

Template definition is syntactically identical to target definition. `template` is optional comma-separated list of templates this template derives from and `SPECIFICATION` may contain the very same things that target node.

Content of `template` node is *not* processed by Bakefile when it is encountered in makefile. It is stored in templates dictionary instead. When a target that derives from the template is encountered, the template is inserted before target's content.

For example consider this makefile fragment:

```
<template id="t1">
  <define>NAME=$(id)</define>
</template>
<template id="t2">
  <include>../headers</include>
</template>

<exe id="app" template="t1,t2">
  <sources>hello.c</sources>
</exe>
```

It looks like this after templates expansion:

```
<exe id="app" template="t1,t2">
  <define>NAME=$(id)</define>
  <include>../headers</include>
  <sources>hello.c</sources>
</exe>
```

using

This commands is used to declare what modules the makefile requires. See more about modules in the section called “Modules”.

```
<using module="MODULE1[,MODULE2[,...]]"/>
```

The effect of `using` is as follows: the modules are added to the list of used modules (unless they are already in it) and additional ruleset files are loaded from Bakefile search paths. Name of every file in every search path is decomposed into components by making every subdirectory name a component and splitting the basename into components by separating it on hyphens. A file is included as soon as all components of its name appear in the list of used modules. The inclusion behaves indentically to include.

Consider this structure of ruleset files:

```
python/common.bakefile        # python,common
python/cxx.bakefile           # python,cxx
cxx-common.bakefile           # cxx,common
```

```
cxx-qt.bakefile               # cxx,qt
qt/python.bakefile            # qt,python
qt/cxx-python.bakefile        # qt,cxx,python
```

Anotated makefile fragment illustrates order of modules loading:

```
<using module="python"/>
<!-- python/common.bakefile loaded -->

<using module="cxx"/>
<!-- cxx-common.bakefile loaded -->
<!-- python/cxx.bakefile loaded -->

<using module="qt"/>
<!-- qt/python.bakefile loaded -->
<!-- cxx-qt.bakefile loaded -->
<!-- qt/cxx-python.bakefile loaded -->
```

(Note that module "common" and module named after the target format are always used. Therefore ruleset files `common/MODULE.bakefile` are always loaded if they exist.)

The command may be used repeatedly in the makefile or included files. Repeating the `using` command with module that was already added to the list of used modules with `using` has no effect.

Parameters:

module

Comma-separated list of modules to use.

In this example the makefile uses Gettext, Python and Pascal modules:

```
<using module="gettext,python"/>
<using module="pascal"/>
```

include

Includes Bakefile file. This is done by loading the file and processing it immediately after `include` command is encountered during parsing. The effect of using `include` is identical to including content of the file in place of the `include` command.

```
<include file="FILENAME" [ignore_missing="0|1"] [once="0|1"]/>
```

Parameters:

file

Name of the file to include. The filename may be either absolute or relative. In the latter case, it is looked up relative to the location of the makefile that contains the `include` command and if that fails, relative to standard Bakefile search paths.

ignore_missing

If set to 1, it is not an error if the file can't be found. If 0, Bakefile will abort with an error if it can't find the file.
Default value: 0

once

If set to 1, then the file won't be included if it was already included previously.
Default value: 0

if

Conditionally process part of the makefile.

```
<if cond="WEAKCONDITION">
  ...statements...
```

```
</if>
```

The condition must be weak. If it evaluates to 1 nodes under `if` node are processed as if they were toplevel nodes. If it evaluates to 0, they are discarded.

fragment

Inserts text into generated native makefile verbatim, so that it is possible to include things not yet supported by Bakefile in the makefiles. The text can be either read from a file or is taken from command node's content. Variables are *not* substituted in fragment's content, it is copied to the makefile as-is, with no changes.

Parameter	**Description**	**Required/Default value**
format	Output format the fragment is for.	required
file	Read the fragment from file.	no file, text is embedded

requires

Declares bakefile's requirements that the installed bakefile version must meet to be able to correctly generate native makefiles from it.

Parameter	**Description**	**Required/Default value**
version	Minimal required version of Bakefile, e.g. `0.1.1`.	optional

Example:

```
<!-- refuse to run with Bakefile < 0.5.0,
     it's missing feature foo: -->
<requires version="0.5.0"/>
```

error

Reports error to output and exits. This command is useful for adding sanity checks to bakefiles (both user bakefiles and format definitions).

```
<!-- This code prevents creation of rules
     for console mode apps: -->
<define-tag name="app-type" rules="exe">
  <if cond="value == 'console'">
    <error>
      Windows CE doesn't support console applications.
      </error>
  </if>
</define-tag>
```

warning

Reports warning to output and exits. This command is useful for adding sanity checks to bakefiles (both user bakefiles and format definitions).

```
<if cond="FORMAT=='msvc'">
    <warning>msvc support is experimental</warning>
</if>
```

echo

Prints the text in tag's value to output and, unlike error, continues processing. This command is useful for debugging bakefiles (e.g. by printing variable values or adding progress messages).

Note that if a variable is used in the text, it must evaluate to a constant (i.e. conditional variables or options cannot be used).

Parameter	Description	Default value
level	Can be `verbose` (in which case the message is printed only when bakefile(1) is run with `--verbose` argument), `debug` (printed only when using the `--debug` flag) or `normal` (message is printed in any case to stdout).	`normal`

Example:

```
<!-- Show the content of the variable X -->
<set var="X">$(someComplexFunction())</set>
<echo>The content of the X variable is: $(X)</echo>
```

Commands for Extending Bakefile

define-rule

Creates a new rule which can then be used as any other rule. A rule consists of the template (which is processed before target-specific code for all targets created by this rule) and unlimited number of define-tag statements that define tags specific to this rule (and derived rules).

The usage of <define-rule> is as follows:

```
<define-rule name="NAME">
  <template>
    <!-- here goes the template for this rule -->
  </template>

  <define-tag name="TAG1">
    ..
  </define-tag>
  <define-tag name="TAG2">
    ..
  </define-tag>
  ...
</define-rule>
```

Parameter	Description	Required/Default value
name	The name of the rule to create.	required
pseudo	Allowed values are `0` and `1`; the value of means that the rule is a pseudotarget.	`0`
extends	A comma-separed list of the rules which are *extended* by this rule. If rule B extends rule A, it means that all tags defined for A are also valid for B and the template of rule B automatically derives from the the template of rule A.	

Example:

```
<!-- Creates a new "copymo" rule with its own specialized
     tags; example usage of this rule:

        <copymo id="i18n">
            <lang>en</lang>
            <mo>myfile.mo</mo>
        </copymo>
-->
<using module="datafiles"/>
<define-rule name="copymo" extends="copy-files">
    <template>
        <srcdir>$(SRCDIR)/locale</srcdir>
        <files>$(__mofiles)</files>
        <dependency-of>all</dependency-of>
    </template>
    <define-tag name="lang">
        <dstdir>$(DATADIR)/locale/$(value)/LC_MESSAGES</dstdir>
    </define-tag>
    <define-tag name="mo">
        <set var="__mofiles">$(value)</set>
    </define-tag>
</define-rule>
```

define-tag

Creates a new tag which can be used inside target definition or rules templates.

This command can be used in two ways: either it's used inside of define-rule, in which case it defines a new tag for the current rule, or it's used in the global scope, in which case it must have the `rules` attribute that specifies which rules the tag applies to.

Parameter	Description	Required/Default value
name	Name of the tag to define.	required
rules	Comma-separed list of rules to which the tag applies.	required in global scope, implicit inside define-rule

Example:

```
<!--
Create a new tag which adds include and lib paths for a "standard"
library and can be used inside <exe> or <dll> tags; e.g.

  <exe id="test">
    <stdlib>lib1</stdlib>
    <stdlib>lib2</stdlib>
  </exe>
-->
<define-tag name="stdlib" rules="exe,dll">
  <include>$(value)/include</include>
  <lib-path>$(value)/lib</lib-path>
</define-tag>
```

define-global-tag

Like define-tag, but creates a tag that can only be used in the global scope (i.e. alongside targets definitions as opposed to inside them).

Parameter	Description	Required/Default value
name	Name of the tag to define.	required

Example:

```
<!--
Create a global tag which defines 3 variables with the same given
prefix and with the same content; e.g.

  <dummyset prefix="test">abc</dummyset>
  <echo>$(test_first) $(test_second) $(test_third)</echo>

will display "abc abc abc"
```

```
-->
<define-global-tag name="dummyset">
  <set var="$(attributes['prefix'])_first">$(value)</set>
  <set var="$(attributes['prefix'])_second">$(value)</set>
  <set var="$(attributes['prefix'])_third">$(value)</set>
</define-global-tag>
```

add-target

Creates a target programmatically.

Using this command is equivalent to defining a target by using the standard rules syntax, but it makes it possible to add a target using dynamically determined rule. As such, it's only useful when implementing other, higher-level rules. This tag is hardly useful for normal uses of bakefile and is used mostly as an internal utility.

This command can only be used inside rule definition, not in the global scope.

Parameter	**Description**	**Required/Default value**
target	ID of the target to create.	required
type	The rule for the target.	required
cond	The condition under which the target is built. In addition to regular condition syntax, two special forms are supported. If the condition is `target`, the condition of the target within which the add-target tag has been used (if any). If the condition has the form of `target and someOtherCondition`, then target's condition as described above will be appended with `and someOtherCondition`.	`1`

Example:

```
<!--
Creates a new EXE target 'myexe'; this is equivalent to
  <exe id="myexe">
    <sources>source1.c</sources>
  </exe>
-->
<add-target target="myexe" type="exe">
  <sources>source1.c</sources>
</add-target>


<!--
Now define a <do_special_cmd> tag which creates a target with
a name dynamically defined by the target from which the tag is used
-->
<define-tag name="do_special_cmd" rules="exe">
  <add-target target="do_special_for_$(id)" type="action" cond="target"/>
  <modify-target target="do_special_for_$(id)">
    <command>special_command $(id)</command>
  </modify-target>
</define-tag>

<exe id="myapp" cond="BUILD_MYAPP=='1'">
  ...
  <!-- the following tag will create a target 'do_special_for_myapp' which will
       be executed only when BUILD_MYAPP=='1' -->
  <do_special_cmd/>
</exe>
```

modify-target

Modifies an existing target by appending tags under this node to its definition.

Parameter	Description	Required/Default value
target	ID of the target to modify.	required

Example:

```
<!-- Modifies the global 'install' target to run an additional command -->
<modify-target target="install">
  <command>$(CP) myfile dest</command>
</modify-target>
```

output

Bakefile uses this command to specify what files a format produces. Output is generated only as the result of `output` command's presence in ruleset.

Parameter	Description	Required/Default value
file	The file where output goes. Commontly used value is `$(OUTPUT_FILE)`.	required
writer	Name of Empy template that is used to generated the output.	required
method	Method of combining generated output with existing content of the file. The default is `replace`, which overwrites the file. `mergeBlocks` divides both the old and the new file's content into blocks that begin with block signature like this: `### beging block BLOCKNAME ###` Blocks of the new content are copied over to the file, replacing old copies of the blocks, but blocks that are not present in new content are preserved. This can be used e.g. to merge configuration settings from several makefiles. `mergeBlocksWithFilelist` works similarly to `mergeBlocks`, but it includes list of input files that generated the block in the output and ensures that blocks that have no generator (e.g. because user's bakefiles changed and no longer cause some piece of code to be generated) are removed from the	`replace`

Parameter	Description	Required/Default value

	output. The list of files is added to block name like this:
	`### beging block`
	
	`BLOCKNAME[file1.bkl,file2.bkl]`
	`###`
	`insertBetweenMarkers` takes first and last line of generated output, finds them in the output file (which must exist and must contain them) and inserts generated content between them.	

Table of Contents

Bakefile rulesets define lots of variables; this chapter provides brief summary of variables that are available for makefile writers.

Format independent variables

Changing Bakefile behaviour

These variables are meant to be changed using `-D` command line argument (see manual page for details).

Variable name	Description	Default
OPTIONS_FILE	If set, then user-configurable part of generated makefile (i.e. options list) is written into another file instead that the makefile will include. `OPTIONS_FILE` value must be relative to `OUTPUT_FILE`.	(not set, empty)
WRITE_OPTIONS_FILE	If `OPTIONS_FILE` is set and `WRITE_OPTIONS_FILE` is 1, then file `OPTIONS_FILE` is created. If 0, then it is not, but it is still included by the main makefile. This allows you to generate options file shared by lots of subproject makefiles and don't repeatedly generate it.	1

Variables for fine-tuning Bakefile's output (rarely needed):

Variable name	Description	Default
VARS_DONT_ELIMINATE	Bakefile normally eliminates all unused variables from the output. In some rare situations, it may not detect that a variable is used, in which case you can tell it to keep the variable by adding its name to this variable. So far this is only useful if the variables is used in makefile code included using fragment.	(empty)
LIB_PAGESIZE	Set this variable to a large power of two if your linker (on Windows) complains that page size is too small when building static library.	`4096`

Directories

Variable name	Description	Default
SRCDIR	Directory to which names of source files are relative to. This value is relative to OUTPUT_FILE (the only exception is the `autoconf` format, which prefixes the value with `$(srcdir)` in order to allow	.

Variable name	Description	Default

	out-of-tree compilation).
	Note that the value of `SRCDIR` *cannot* be set manually; if you want to change it, you must use the **set-srcdir** command as early in your bakefile as possible. For example:
	`<makefile>`
	`  <set-srcdir>../..</set-srcdir>`
	`  ..`
	`</makefile>`
	The argument to **set-srcdir** must be a constant expression.	

BUILDDIR	Directory where object files and executables are built. This value is relative to OUTPUT_FILE.	(depends on format)

Installation Directories

These are standard installation directories as used on Unix (most notably in Autoconf). They are used by **install** target if the backend supports it. They are defined on all platforms. You can change their values freely (unless you are using Autoconf backend).

Variable name	Description	Default
PREFIX	Base directory for installed files.	/usr/local on Unix
BINDIR	Directory where programs are installed.	$(PREFIX)/bin on Unix
LIBDIR	Directory where libraries are installed.	$(PREFIX)/lib on Unix
DLLDIR	Directory where DLLs are installed.	$(PREFIX)/lib on Unix, $(PREFIX)/bin when targetting win32
INCLUDEDIR	Directory where C and C++ headers are installed.	$(PREFIX)/include on Unix
DATADIR	Directory where data files are installed.	$(PREFIX)/share on Unix

Recognizing Platform

All of these are variables defined to either 0 or 1, with the exception of `autoconf` format backend where they are options.

Variable name	Description
PLATFORM_UNIX	UNIX variant
PLATFORM_WIN32	32bit Windows
PLATFORM_MSDOS	MS-DOS
PLATFORM_MAC	Mac OS X or Mac Classic
PLATFORM_MACOSX	Mac OS X
PLATFORM_OS2	OS/2
PLATFORM_BEOS	BeOS
PLATFORM_SYMBIAN	Symbian OS

Format features

Variable name	Description
FORMAT_HAS_VARIABLES	This boolean flag indicates whether the output format supports variables. If it does, then some space and time optimization are possible and long Bakefile variables that would otherwise be expanded into literals are left as variables in generated makefile/project.
FORMAT_SUPPORTS_CONDITIONS	Whether the output format can handle conditions (i.e. variable values depending on conditions and conditionally built targets) at all. Most make-tools do, but e.g. MSVC project files can't do it. If set to 0, then the targets and variables are "flattened", i.e. expanded into multiple 'configurations' as in many IDEs.
FORMAT_SUPPORTS_CONFIGURATIONS	Whether it is at least possible to have multiple configurations if conditions are not supported (such as in IDEs). Meaningless if FORMAT_SUPPORTS_CONDITIONS=1. If both FORMAT_SUPPORTS_CONDITIONS and FORMAT_SUPPORTS_CONFIGURATIONS are 0, then we're in deep trouble and we can only generate makefiles that are not configurable.
FORMAT_SUPPORTS_ACTIONS	Whether the output format can handle actions at all. Most make-tools do, but IDE project files typically don't.
FORMAT_SUPPORTS_SUBPROJECTS	Whether the output format can handle subprojects at all. Most make-tools do, but IDE project files typically don't.
FORMAT_NEEDS_OPTION_VALUES_FOR_CONDITIONS	Whether the output format needs to have options that are used by conditions listed in the output file. This is true in majority of cases because the conditions take form such as `"!if $(OPT) == value"` and OPT must be defined, but there's one exception: autoconf. It decides on whether the condition is true or false in autoconf_inc.m4 and Makefile.in does not need the variables, so we can safely purge them and save some space.

Miscellaneous

Variable name	Description
BAKEFILE_VERSION	String with Bakefile version number. The version is formed from three numbers delimined by period. Read only.
OPTIONS	Space-separation list of options defined in the makefiles. Note that the value of this variable changes during processing as new options are defined!
INPUT_FILE	Name of input file. The name is always absolute path. Read only.
INPUT_FILE_ARG	Same as INPUT_FILE, but the name is in exactly

Variable name	Description
	same form as it was passed on command line, it's not made absolute as in case of `INPUT_FILE`.
OUTPUT_FILE	Name of the file where generated native makefile will be written. Read only.
FORMAT	Format of makefile currently being generated, e.g. `autoconf`. Always constant expression.
COMPILER	Short identifier of used compiler (e.g. "bcc" or "vc6"). This variable is only defined for Windows compilers and is guaranteed to evaluate to constant expression.
TOOLSET	What kind of tools the makefile uses. Use this to determine what commands to put into command tags. Can be one of `win32`, `unix`. Always a constanst expression.
EOL_STYLE	Default line endings style for current format, one of `unix`, `dos` and `mac`.
LF	Line feed character (\n in C).
TAB	Tabelator character (\t in C).
DOLLAR	Dollar sign ($).
SPACE	Space character (" "). Note that `SPACE` is not evaluated and so you can't use it in places where constant expression is expected.
DIRSEP	Character used to separate directory components in paths on target platform (/ on Unix, \ on Windows).

Standard makefile variables

Makefile-based formats (`gnu`, `msvc` etc.) define standard options `CC`, `CXX`, `CFLAGS`, `CXXFLAGS`, `CPPFLAGS`, `LDFLAGS` for specifying the compiler and its flags. Their default values are set to the default or typical compiler. If necessary, the defaults for `FOO` can be overriden by setting the `DEFAULT_FOO` (e.g. `DEFAULT_CXX` for the C++ compiler) variable to a constant value anywhere in user bakefiles.

Format specific variables

autoconf

Variable name	Description	Default
AUTOCONF_MACROS_FILE	Where `configure.in` macros for setting options and conditional variables (mostly `AC_SUBST` calls) are written. This file *must* be included by your `configure.in` script. Set it to empty string to disable creation of this file (e.g. for subprojects of main project, see also `OPTIONS_FILE`). Note that the value of `AUTOCONF_MACROS_FILE` *shouldn't* be set manually; if you want to change it, use the **autoconf-needs-**	autoconf_inc.m4

Variable name	Description	Default
	macro command. For example: `<makefile>` `<autoconf-needs-macro>AC_BAKEFILE_PYTHON</autoconf-needs-macro>` `..` `</makefile>` The argument to **autoconf-needs-macro** must be a constant expression. It can be repeated more than once.	

dmars, dmars_smake

Variable name	Description	Default
DMARS_MEM_POOL_SIZE	Specifies size of memory pool used by the **dmc** compiler. The default should be sufficient most of the time and only needs increasing if the compiler fails to compile source code with out of memory errors.	99

msvs2005prj, msvs2008prj

Variable name	Description	Default
MSVS_PLATFORMS	Comma-separated list of platforms to generate project configurations for. Can be set by the user using the `-D` command line argument. See MS VisualStudio 2005/2008 format documentation for details.	win32
MSVS_PLATFORM	This variable is not meant to be set by the user. It's an option created by the format itself. It's allowed values are values from MSVS_PLATFORMS variable set by the user. It can be used by the bakefiles to generate different project settings for different platform, in the same way user-added options are used.	
MSVS_PROJECT_FILE	Unlike other variables in this section, this variable can optionally be set *on targets* to specify the location of `.vcproj` file created for them. The location is relative to OUTPUT_FILE. See an example: `<exe id="hello_world">` `...` `<!-- create subdirectory for projects if needed and put it there -->`	(derived from OUTPUT_FILE)

Variable name	Description	Default
	`<set var="MSVS_PROJECT_FILE">examples/HelloWorld.vcproj</set> </exe>`	

msvs2003prj

The MSVS_PLATFORM option is present in this format as well, but it is always set to `win32`. MSVS_PROJECT_FILE is fully supported.

Table of Contents

Introduction

In Bakefile, the expression inside `$(...)` doesn't have to be a variable name, it can be *any* Python expression. Expressions that cannot be evaluated at runtime and are translated into output format conditions are more limited, but as long as the expression is evaluated only at Bakefile execution time, it can be any valid Python expression. In particular, any Python functions may be called.

In order to make common tasks easier, Bakefile provides miscellaneous utility functions which can be used in your bakefiles. Unlike tags and rules provided through modules, the functions documented in this section are available everywhere in Bakefile.

Except where explicitely stated differently, all functions accept as arguments Python strings.

How to use a Python function in a bakefile

Python instructions and thus also Python calls to functions, can be used in bakefile wrapping them into the `$( )` symbols. E.g. suppose you want to use the fileList function described below to set variable `A`; you should then write:

```
<set var="A">
    $(fileList('mypath/*.c'))
</set>
```

Python functions

The following Python functions are defined:

envvar

`envvar("name")`
Returns reference to environment variable `name`. This function should be used instead of `$(DOLLAR)(name)` idiom, because some output formats (namely, Watcom makefiles) use different syntax for referencing environment variables.

isconst

`isconst(expr)`
Returns `true` if the *expression* (i.e. not variable name as in the case of isdefined etc.) given as argument is constant expression.

isdefined

`isdefined(name)`
Returns `true` if the given string is the name of an option or a (conditional) variable previously defined in the bakefile.

isoption

`isoption(name)`
Returns `true` if the given string is the name of an option previously defined in the bakefile.

iscondvar

`iscondvar(name)`
Returns `true` if the given string is the name of a conditional variable previously defined in the bakefile.

ifthenelse

`ifthenelse(cond, iftrue, iffalse)`
Allows to write if-then-else constructs inside bakefiles. The arguments are respectively the *if condition* (use Python syntax!), the Python expression to execute in case the condition results true and the Python expression to execute in case the condition results false.

ref

`ref(var, target=None)`
Creates a reference to the given `var` variable which will be evaluated only in the final stage of bakefile processing.

isDeadTarget

`isDeadTarget(target)`
Returns *true* if the given string is the name of a conditional target whose condition is never met.

substituteFromDict

`substituteFromDict(var, dict, desc=None)`
Returns the *value* of the dictionary entry whose *key* matches the value of the variable or option named `var`. E.g.

```
<set var="A">
    $(substituteFromDict(OPTION,{'1':'value1','0':'value0'}))
</set>
```

sets *A* to *value1* if *OPTION* is *1* or to *value2* if *OPTION* is *0*. Note that Python requires curly brackets to define a dictionary.

nativePaths

`nativePaths(filenames)`
Returns the given string with the `/` characters substituted by the content of the DIRSEP variable (see the Variables section).

addPrefixIfNotEmpty

`addPrefixIfNotEmpty(prefix, value)`
Returns the `value` string prefixed with `prefix`, unless `value` is empty.

addPrefixToList

`addPrefixToList(prefix, value)`
Adds `prefix` to every item in `value` interpreted as whitespace-separated list. E.g.

```
<set var="A">
    $(addPrefixToList('file','1.txt 2.txt 3.txt'))
</set>
```

sets the *A* variable to `file1.txt file2.txt file3.txt`

safeSplit

`safeSplit(str)`
Splits the given string like the built-in split() Python function but, unlike the Python `split()` function, recognizes that an expression like

```
"$(myPythonFuncCall(arg1, arg2)) item2"
```

must be split as
```
[ "$(myPythonFuncCall(arg1, arg2))", "item2" ]
```

and not as the built-in split() function would do
```
[ "$(myPythonFuncCall(arg1,", "arg2))", "item2" ]
```

fileList

`fileList(path)`
Returns a string containing a space-separed list of all files found in the given `path`. `path` typically is a relative path (absolute paths should be avoided in well-designed bakefiles) with a mask used to match only wanted files.

When the given path is relative, it must be relative to the SRCDIR global variable; remember that SRCDIR is in turn relative to the location of the generated makefile (see OUTPUT_FILE).

Additionally this function can accept Python lists of strings, too. The returned value is the list of all files found in all the paths of the list. E.g.
```
<sources>$(fileList('../src/*.cpp'))</sources>
<sources>$(fileList(['../src/*.cpp', '../src/*.c']))</sources>
```

removeDuplicates

`removeDuplicates(list)`
Returns a copy of the given (space-separed) list with all duplicate tokens removed.

Table of Contents

Depending on the format there are some additional steps required to get extra functionality specific to this format.

MS VisualStudio 2005/2008 extended functionality

The `msvs2005prj` and `msvs2008prj` formats are capable of generating project files both for Win32 PC platform and for embedded platform. By default, only Win32 configurations are generated, embedded configurations must be explicitly enabled.

This is done by setting the `MSVS_PLATFORMS` variable to comma-separated list of platforms to use. Accepted platform identifiers are:

Platform identifier	**Visual Studio name**
`win32`	Win32
`win64`	x64
`pocketpc2003`	Pocket PC 2003 (ARMV4)

The variable must be set before user bakefile file is processed, i.e. it has to be done using command-line `-D` argument or in the `Bakefiles.bkgen` file.

For example, the following command causes Bakefile to generate projects for Pocket PC 2003:

```
$        bakefile -fmsvs2005prj -DMSVS_PLATFORMS=pocketpc2003 hello.bkl
```

And this generates hybrid project for both PocketPC and Win32 platforms:

```
$        bakefile -fmsvs2005prj -DMSVS_PLATFORMS=win32,pocketpc2003 hello.bkl
```

Watcom format extended functionality

Open Watcom compiler has possibility of crossplatform building for many platforms. MS Windows binaries are the most common used output of this compiler but it has also possibility of building for DOS, OS/2 and other operating systems. Because of that by default watcom makefiles are outputed with settings for building windows binaries. This default setting can be changed by defining additional platform variable.

```
# generate makefile.wat dedicated to windows development
bakefile -f watcom file.bkl

# generate makefile.wat dedicated to DOS development in extended 32-bit mode
bakefile -f watcom -DPLATFORM_MSDOS=1 file.bkl

# generate makefile.wat dedicated to OS/2 development
bakefile -f watcom -DPLATFORM_OS2=1 file.bkl
```

PLATFORM_MSDOS note

Only 32-bit DOS mode with dedicated extender is supported. Generated makefile contains additional `DOS32` variable which points to desired extender. Expected values: `X32VM`, `X32`, `PMODEW`, `CAUSEWAY`, `DOS32A` and default `DOS4GW`.

Install and uninstall support on Windows

By default the install-to and install-headers-to tags won't have any effect on Windows since the install and uninstall MAKE targets are not very common there (unlike in the Unix world).

However you may find it useful to have (un)installation targets under Windows as well. In this case you can enable this feature by defining the `FORMAT_HAS_MAKE_INSTALL` variable and setting it to `1`:

```
# enable install and uninstall support also under Windows:
```

```
bakefile file.bkl -f msvc -DFORMAT_HAS_MAKE_INSTALL=1
```

When enabling the install/uninstall support for Windows, you should also set the desired `PREFIX` and `EXEC_PREFIX` in your bakefile:

```
<set var="PREFIX">%MYPROJECTROOT%</set>
<set var="EXEC_PREFIX">%MYPROJECTROOT%</set>
```

Table of Contents

datafiles

This module provides rules for installing data files during `make install` phase. It also defines two rules for copying files during build process (typically from source to build directory).

Implementation Note

This module currently works only with the `autoconf` format.

The following targets are defined in `datafiles` module:

data-files

Pseudo target that declares installable data files. Note that *all* files are installed into target directory, their relative directories are *not* preserved.

Tag	**Description**
srcdir	Directory where source files are.
files	List of files to copy. Names are relative to source directory. May be used more than once.
install-to	Directory where to install the files.

data-files-ng

Same as data-files, except that **data-files-ng** is real target and not pseudo target. This has two consequences: the target must have `id` set and it can be conditional.

script-files

Same as data-files, but installed files have executable permissions on Unix.

script-files-ng

Same as data-files-ng, but installed files have executable permissions on Unix.

data-files-tree

Unlike data-files, this rule preserves directory structure of installed files. Available tags are same and have same meaning as data-files tags.

copy-files

Copies file(s) from source directory to destination directory. Creates destionation directory if it doesn't exist.

Tag	**Description**
srcdir	Directory where source files are. This tag is not required (unlike the other two) -- source directory is empty by default.
files	List of files to copy. Names are relative to source directory.
dstdir	Directory where to copy the files.

copy-file-to-file

Copies single file to another file.

Tag	Description
src	Source file.
dst	Destination file.

mkdir

Creates directory.

Tag	Description
dir	Name of the directory.

pkgconfig

This module provides a simple rule for installing and uninstalling pkg-config template files.

Implementation Note

This module currently works only with the `autoconf` format because pkg-config files contain values such as prefix that are set by configure.

The following targets are defined in `pkgconfig` module:

pkgconfig

Installs a .pc template file in the standard location of the pkgconfig files (i.e. LIBDIR/pkgconfig).

Tag	Description
src	Name of the pkgconfig file to install. Can contain a relative path. E.g. `build/myprj.pc`

Table of Contents

Introduction

As soon as you start using Bakefile for your project and you need to generate many makefile formats from your bakefiles (after all, this is the purpose of Bakefile!), you'll find very useful to automate the regeneration process.

Here is where bakefile_gen(1) comes into play. You can script all the bakefile calls you would have to do manually in a single `Bakefiles.bkgen` file and then just call: `bakefile_gen` to run Bakefile for all the formats you need to regenerate on all the bakefiles which are required by your project.

`Bakefiles.bkgen` files use a simple XML format to describe what outputs and how to generate. The root tag is called `bakefile-gen` and inside it you can use any of the tags described below.

bakefile_gen tags

In addition to the following tags, bakefile_gen also supports the include tag.

Tag

Description

input

Adds the given list of whitespace-separed bakefiles to the list of bakefiles which must be regenerated.

You can use wildcards and relative paths to match all the bakefiles scattered in the directory tree of your project. Example:

```
<!-- tell bakefile_gen to regenerate all the bakefiles of this project -->
<input>
    mybakefile1.bkl
    ../mybakefile2.bkl
    ../../../build/bakefiles/*.bkl
</input>
```

add-formats

Adds the comma-separed list of formats contained in this tag, to the list of formats to regenerate.

You can use the *files* attribute of this tag to selectively add the listed formats to a (set of) bakefile(s) only.

```
<!-- add the GNU format to all bakefiles under 'build/bakefiles' -->
<add-formats files="build/bakefiles/*.bkl">gnu</add-formats>
```

del-formats

Removes the comma-separed list of formats contained in this tag, from the list of formats to regenerate.

You can use the `files` attribute of this tag to selectively add the listed formats to a (set of) bakefile(s) only.

```
<!-- remove some rarely used formats from the bakefiles under the 'a' and 'b' subfolders
-->
<del-formats
files="a/*.bkl,b/*.bkl">cbuilderx,dmars,dmars_smake,msevc4prj,symbian,xcode2</disable-
formats>
```

enable-formats

Enables the regeneration of the comma-separed list of formats contained in this tag.

Note that by default all formats supported by Bakefile are enabled, thus this tag will actually have some effect only if you used the disable-formats tag before.

```
<disable-formats>msvc,gnu</disable-formats>
...
<!-- we've changed idea; turn GNU format on -->
<enable-formats>gnu</enable-formats>
```

disable-formats

Disables the regeneration of the comma-separed list of formats contained in this tag.

Typically you'll want to use this tag to disable all those formats you are not interested to.

```
<!-- disable rarely used formats: -->
<disable-formats>cbuilderx,dmars,dmars_smake,msevc4prj,symbian,xcode2</disable-formats>
```

add-flags

Adds some flags to the command executed to regenerate the bakefiles.

You can use the `files` attribute of this tag to selectively add the flags to a (set of) bakefile(s) only.

You can use the `formats` attribute of this tag to selectively add the flags to a (set of) format(s) only.

Additionally, inside this tag, bakefile_gen recognizes various variables: `$(INPUT_FILE)` is the path of the bakefile being processed; `$(INPUT_FILE_BASENAME)` is the filename of the bakefile being processed; `$(INPUT_FILE_BASENAME_NOEXT)` is the filename of the bakefile being processed without the extension; `$(INPUT_FILE_DIR)` is the directory of the bakefile being processed.

```
<!-- tell bakefile to output the generated Makefile.in for bake.bkl two levels up -->
<add-flags files="bake.bkl" formats="autoconf">
    -o../../Makefile.in
</add-flags>

<!-- always generate the windows makefiles in ../msw respect the bakefile's being
processed: -->
<add-flags formats="borland">-o$(INPUT_FILE_DIR)/../msw/makefile.bcc</add-flags>
<add-flags formats="mingw">-o$(INPUT_FILE_DIR)/../msw/makefile.gcc</add-flags>
<add-flags formats="msvc">-o$(INPUT_FILE_DIR)/../msw/makefile.vc</add-flags>
<add-flags formats="watcom">-o$(INPUT_FILE_DIR)/../msw/makefile.wat</add-flags>
```

del-flags

Removes some flags to the command executed to regenerate the bakefiles.

You can use the `files` attribute of this tag to selectively remove the flags to a (set of) bakefile(s) only.

You can use the `formats` attribute of this tag to selectively remove the flags to a (set of) format(s) only.

```
<add-flags>-DVARIABLE1=value</add-flags>

<!-- delete the -DVARIABLE1=value option from the MSVC and BORLAND formats -->
<del-flags formats="msvc,borland">
    -DVARIABLE1=value
</del-flags>
```

Processing order

`Bakefiles.bkgen` file is processed in the following order:

1. `disable-formats` entries are read into blacklist of formats to globally ignore
2. `enable-formats` entries are read and the formats listed are *removed* from the blacklist (so that your `Bakefiles.local.bkgen` file can re-enable something disabled by default).
3. `add-formats` and `del-formats` are processed in the order they appear in the file. They specify which formats should be generated for which files (the default being all files), assuming the blacklist is empty (in other words, they describe what this `Bakefiles.bkgen` is *capable* of generating).
4. The list from step 3. is filtered using the blacklist from steps 1. and 2.

Table of Contents

Name

bakefile — native makefiles generator

Synopsis

`bakefile` [`--version`] [`--help`] `-fFORMAT` `-oOUTFILE` [`--eol=[format|dos|unix|mac|native]`] [`--wrap-output=[no|LENGTH]`] [`-DVAR=VALUE` ...] [`-IPATH` ...] [`-v`] [`-q`] [`--dry-run`] [`--touch`] [`--dump`] `file.bkl`

Description

bakefile creates various types of Makefiles and project files from a single project description called a "Bakefile".

Command Line Options

`--version`

Display Bakefile version and exit.

`-h`, `--help`

Display usage information and exit.

`-fFORMAT`, `--format=FORMAT`

Specify output format. Bakefile supports the following formats:

Format	File(s) Generated
autoconf	Makefile.in for GNU Autoconf
borland	Makefile for Borland C++ and Borland make
dmars	Generic Makefile for Digital Mars C/C++
dmars_smake	Makefile for Digital Mars C/C++ with SMAKE
gnu	Makefile for GNU toolchain: GNU Make, GCC, etc.
mingw	Makefile for MinGW toolchain: mingw32-make, MinGW port of GCC, etc.
msvc	Makefile for Visual C++ with Microsoft nmake
msvc6prj	Microsoft Visual C++ 6.0 project files
msevc4prj	Microsoft Embedded Visual C++ 4 project files
msvs2003prj	MS Visual Studio 2003 project files
msvs2005prj	MS Visual Studio 2005 project files
msvs2008prj	MS Visual Studio 2008 project files
suncc	GNU makefile for SunCC compiler
symbian	Symbian development files
watcom	Makefile for OpenWatcom C/C++
xcode2	Apple Xcode 2.4 project files

`-oOUTFILE`, `--output=OUTFILE`

File to write generated makefile to. For those backends that generate more than one file, this option specifies the name of the main makefile.
This option has special meaning for msvs200xprj formats: by default, both the project files (one for each

target) and a solution file, containing all the project files, are generated. However if OUTFILE is a file with .vcproj extension, then only the (necessarily unique) project file will be generated.

`-DVAR=VALUE`

Define Bakefile variable. This definition overrides any definition from the ruleset or input makefile. You can use it to customize generated output.

`--eol=[format|dos|unix|mac|native]`

Change the type of line endings used by general files. `dos`, `unix` and `mac` specify the line endings used by respective platforms. `native` will use line endings of the platform Bakefile is ran on (doing this is usually a bad idea, but it is useful e.g. when checking generated files into RCS system that can't deal with line endings correctly, such as CVS). The default value is `format` and means that the most appropriate line endings for the output format will be used - Windows makefiles will use DOS line endings, Autoconf makefiles will use Unix ones and so on.

`--wrap-output=[no|LENGTH]`

Change line wrappings behavior. By default, Bakefile wraps generated makefiles so that lines don't exceed 75 characters. Use this option to either change the limit or to disable wrapping entirely by using `no` as the value.

`-IPATH`

Add path to the list of directories where Bakefile looks for rules and output templates.

`--dry-run`

Process the bakefile normally, but instead of creating or modifying files, just print which files would be changed without actually modifying them.

`-v`, `--verbose`

Be verbose.

`-q`, `--quiet`

Supress all output except for errors.

`--touch`

Always touch output files, even if their content doesn't change.

`--debug`

show internal debugging information

`--dump`

Dump all Bakefile variables and targets to standard output instead of generating output. This is only useful for debugging Bakefile or ill-behaving makefiles.

`--output-deps=FILE`

Output dependency information needed by **bakefile_gen** utility

`--output-changes=FILE`

Store list of changed files to the given file

`--xml-cache=FILE`

specify cache file where **bakefile_gen** stores pre-parsed XML files

Environment Variables

BAKEFILE_PATHS

List of directories where ruleset files are looked for (syntax is same as in `PATH`). Bakefile's data directory is always searched in addition to paths listed in `BAKEFILE_PATHS`, but `BAKEFILE_PATHS` has higher priority.

Name

bakefile_gen — batch bakefile generation

Synopsis

`bakefile_gen` [`-dDESCFILE` ...] [`-FFORMATS` ...] [`-DVAR=VALUE` ...] [`-IPATH` ...] [`-c`] [`--list-files`] [`-j`] [`-p`] [`--dry-run`] [`-k`] [`-n`] [`-v`] [`-V`]

Description

Calls **bakefile** with flags listed in description file (`Bakefiles.bkgen` or file specified using the --desc option).

Command Line Options

`--desc=DESCFILE`

Uses given description file instead of `Bakefiles.bkgen`.

`--formats=FORMATS`

Calls Bakefile to generate only makefiles for specified formats. `FORMATS` is comma-separed list of format names. Formats not included in the list are ignored even if they are listed in <add-formats> tags in the description file.

`--bakefiles=BAKEFILES`

Calls Bakefile to generate makefiles only from bakefiles that match any wildcard in comma-separed list of wildcards in `BAKEFILES`. Input files specified using the <input> tag in the description file that don't match any of the wildcards are not processed.

`-DVAR=VALUE`

Pass variable definition to Bakefile, overriding any definition in description file or the input bakefile.

`-IPATH`

Add path to the list of directories where Bakefile looks for rules and output templates.

`-c`, `--clean`

Removes all output files instead of generating them.

`--list-files`

Prints the list of output files that would be generated instead of creating them. This command respects the constraints specified using the `--format` and `--bakefile` options. It can be used for example to create the list of all makefiles for given format.

`-j`, `--jobs`

Number of jobs to run simultaneously. Default is the number of CPUs.

`-p`, `--pretend`

Don't do anything, only display actions that would be performed.

`--dry-run`

Process the bakefile normally, but instead of creating or modifying files, just print which files would be changed without actually modifying them.

`-k`, `--keep-going`

Do not stop when a target fails.

`-B`, `--always-make`

Pretend that all makefiles are out of date and regenerate all of them. `-f` and `-b` options are respected.

`-v`, `--verbose`

Display detailed information.

`-V`, `--very-verbose`

Display even more detailed information.

`--help`

Display usage information for **bakefile_gen**

Name

bakefilize — prepare Bakefile project for use with Autoconf

Synopsis

`bakefilize` [`--copy`] [`--dry-run`] [`--force`] [`--verbose`] [`--help`]

Description

For the "autoconf" format, Bakefile creates Makefile.in files that depend on the availability of common pieces of a GNU build system. (`config.guess`, `install-sh`, etc.) These tools are part of Automake, which can copy these files into a project's directory during processing. **bakefilize** effectively replaces the **automake --add-missing** feature.

It is standard practice in Autoconf-based projects to provide a "bootstrap" script (commonly called either **bootstrap** or **autogen.sh**) to run commands like **autoconf** with the proper flags and in the proper order. You should run **bakefilize** in that script, at some point before the **configure** script runs.

Command Line Options

`-c`, `--copy`

Copy the files from the Automake directory, rather than the default behavior of making symbolic links.

`-n`, `--dry-run`

Only show the commands that would be executed.

`-f`, `--force`

Replace all existing files, instead of only adding missing files.

`-v`, `--verbose`

Show debugging messages.

`--help`

Display usage information for **bakefilize**