Bakefile

Release 1.2.5.1-73

December 24, 2018

Contents

Contents:

CHAPTER 1

Introduction

1.1 Just Show Me How It Works

If you already have an idea of what Bakefile does, please feel free to skip directly to the *Tutorial*.

1.2 Motivation

Almost any introductory C or C++ book starts with presenting a version of "Hello, world" program. Some of them also show how to compile the program, although often enough the reader is advised to refer to the documentation of the compiler to learn how to do it. And almost none of the books or tutorials address the issue of how can the program be compiled not in a single environment and with a single compiler but on several different platforms or, perhaps, with different compilers on the same platform. And there is a good reason for avoiding discussing this: surprisingly, 40 years after the invention of C programming language, there is still no satisfactory solution to this problem.

Before discussing how Bakefile helps to address it, let's briefly mention the two usual anti-solutions. The first one of them is to naively separately write build scripts, make files or projects for each of the environments used. While there is no problem with this approach as long as a single such environment is used, accurately propagating changes to one of the files to all the other ones becomes bothersome even when there are only two of them and, in practice, the divergences almost inevitably appear. For a cross-platform project, targeting many different platform, this quickly becomes a serious maintenance burden. But even for projects used under a single platform, keeping different versions of the project files can be annoying (a typical example is that of Microsoft Visual C++ compiler which has changed its project file format with each and every one of its last 5 releases).

The problems of this naive approach have resulted in creation of a huge number of cross-platform build tools. This manual is too short to discuss the cons and pros of each of them, but they all share one fundamental weakness that makes us classify them all as an anti-solution: the use of such tool replaces the use of the normal tools used for software development under any given platform. As anybody who has tried to convince a seasoned Unix programmer to switch to using Xcode instead of make, a Windows developer to use make instead of Microsoft Visual Studio or someone used to work with Xcode to live without it can attest, this can be a bad idea with effects ranging from the drop of participation in an Open Source project to poisoning the atmosphere in the office in a company one.

1.3 The Bakefile Solution

The solution proposed by Bakefile allows to combine the use of native development tools with the lack of maintenance headaches due to having to modify the files for each of the tool chains manually. Bakefile is a meta-make tool which does *not* replace the normal make program or IDE used for building the project but simply generates the input files for it from a unique meta-makefile which we call a "bakefile".

It is important to notice that the make and project files created by bakefile*[0] are as similar to the files with the same functionality created by hand as possible, i.e. are easy to understand for anybody familiar with the underlying tool. These files still should not be modified directly but, in a pinch, this can be done – and is as easy to do as with manually crafted versions.

This need to restrict modifications to the bakefiles themselves and avoid modifying the files generated from them does require some discipline but this is the smallest possible price to pay for solving the problem of how to let developers use their favourite tool chain under all platforms without having to manually ensure the synchronization between all of them.

[0] This process is, inevitably, called "baking".

CHAPTER 2

Tutorial

2.1 Hello, Bakefile

After complaining about the lack of examples of how "Hello, World" program can be built in a portable way in the introduction, we'd be amiss to not provide an example of doing this here. So, assuming the text of the program is in the file `hello.c`, here is the corresponding `hello.bkl` bakefile to build it:

```
toolsets = gnu vs2010;
program hello {
    sources { hello.cpp } // could also have been "hello.c"
}
```

To produce something interesting from it we need to run bakefile simply passing it this file name as argument:

```
$ bkl hello.bkl
```

or maybe:

```
C:\> bkl hello.bkl
```

In the grand tradition of Unix tools, Bakefile doesn't produce any output after running successfully. If this is too closemouthed for your taste, you can try adding `-v` option which will indicate all the files read and, more importantly, written by the tool. Using this option or just examining the directory you ran Bakefile in you can see that it created several new files.

In our case, they will be `GNUmakefile` for "gnu" toolset (the one using standard GNU development tools) or `hello.sln` and its supporting files for "vs2010" toolset (Microsoft Visual Studio 2010) one. These files then can be used to build the project in the usual way, i.e. by running "make" or opening the file in the IDE.

Please check that you can run Bakefile and generate a working make or project file appropriate for your platform ("gnu" for Unix systems including OS X, "vs2010" for Microsoft Windows ones).

2.2 Example Explained

Bakefile input language is C-like, with braces and semicolons playing their usual roles. Additionally, both C and C++ style comments can be used. It however also borrows some simple and uncontroversial elements*[0] of make syntax. Notably, there is no need to quote literal strings and, because of this, Bakefile variables – such as `toolsets` above – need to be explicitly dereferenced using make-file `$(toolsets)` expression when needed.

Knowing this we can see that the first line of the hello bakefile above simply sets a variable `toolsets` to the list of two strings. This variable is special for Bakefile – and hence is called "property" rather than a simple "variable" – and indicates which make or project files should be generated when it is ran. It must be present in all bakefiles as no output would be created without it.

The next block – `program hello { ... }` – defines an executable target with the name "hello". All the declarations until the closing bracket affect this target only. In this simple example the only thing that we have here is the definition of the sources which should be compiled to build the target. Source files can be of any type but currently Bakefile only handles C (extension `.c`) and C++ (extension `.cpp`, `.cxx` or `.C`) files automatically and custom compilation rules need to be defined for the other ones. In any real project there are going to be more than one source file, of course. In this case they can just be listed all together inside the same block like this:

```
sources {
    foo.cpp subdir/bar.cpp
    // Not necessarily on the same line
    baz.cpp
}
```

or they can be separated in several blocks:

```
sources { foo.cpp }
sources { subdir/bar.cpp }
sources { baz.cpp }
```

The two fragments have exactly the same meaning.

2.3 Customizing Your Project

2.3.1 Compilation And Linking Options

A realistic project needs to specify not only the list of files to compile but also the options to use for compiling them. The most common options needed for C/C++ programs are the include paths allowing to find the header file used and the preprocessor definitions and Bakefile provides convenient way to set both of them by assigning to `defines` and `includedirs` properties inside a target:

```
program hello {
    defines = FOO "DEBUG=1";    // Quotes are needed if value is given
    includedirs = ../include;
    ...
}
```

These properties can be set more than once but each subsequent assignment overrides the previous value which is not particular useful. It can be more helpful to append another value to the property instead, for example:

[0] So no meaningful tabulations or backslashes for line continuation.

```
program hello {
    defines = FOO;
    ...
    defines += BAR;
    ...
    defines += "VERSION=17";
    defines += "VERSION_STR=\"v17\"";
}
```

will define "FOO" and "BAR" symbols (without value) as well as "VERSION" with the value of 17 and "VERSION_STR" with the value as a C string during compilation. This is still not very exciting as all these values could have been set at once, but the possibility of conditional assignment is more interesting:

```
program hello {
    if ( $(toolset) == gnu )
        defines += LINUX;
    if ( $(toolset) == vs2010 )
        defines += MSW;
}
```

would define `LINUX` only for makefile-based build and `MSW` for the project files.

While `defines` and `includedirs` are usually enough to cover 90% of your needs, sometimes some other compiler options may need to be specified and the `compiler-options` property can be used for this: you can simply any options you want to be passed to C or C++ compiler into it. If you need to be more precise and only use some options with a particular compiler in a project using more than one of them, you can also use `c-compiler-options` or `cxx-compiler-options`.

One aspect of using these properties is that different compilers use different format for their options, so it's usually impossible to use the same value for all of them. Moreover, sometimes you may actually need to use custom options for a single toolset only. This can be done by explicitly testing for the toolset being used. For example, to use C++ 11 features with GNU compiler you could do

```
if ( $(toolset) == gnu )
    cxx-compiler-options = "-std=c++11"
```

Similarly, any non trivial project usually links with some external libraries. To specify these libraries, you need to assign to the `libs` property and also may need to set `libdirs` if these libraries are not present in the standard search path:

```
program hello {
    libdirs = ../3rdparty/somelib/lib;
    libs = somelib;
}
```

Notice that you only need to do the latter if the libraries are not built as part of the same project, otherwise you should simply list them as dependencies as shown in the next section.

And if you need to use any other linker option you can specify it using `link-options` property. As with compiler options, you would normally test for the toolkit before doing it as linker options are toolset-specific.

2.3.2 Multiple Modules

Larger projects typically consist in more than just a single executable but of several of them and some libraries. The same bakefile can contain definitions of all of them:

```
library network {
    sources { ... }
}

library gui {
    sources { ... }
}

program main {
    deps = network gui;
    ...
}
```

In this case, the libraries will be built before the main executable and will be linked with it. Bakefile is smart about linking and if a library has dependencies of its own, these will be linked in as well.

Alternatively, you can define each library or executable in its own bakefile. This is especially convenient if each of them is built in a separate directory. In this case you can use `submodule` keyword to include the sub-bakefiles.

2.3.3 Assorted Other Options

Under Windows, console and GUI programs are compiled differently. By default, Bakefile builds console executables. You can change this by setting the `win32-subsystem` property to `windows`.

Another Windows-specific peculiarity is that standard C run-time library headers as well as Platform SDK headers are compiled differently depending on whether `_UNICODE` and `UNICODE` macros, respectively, are defined or not. By default, Bakefile does define these macros but you can set `win32-unicode` target property to `false` to prevent it from doing it.

Finally, Windows projects generated by default only contain configurations for Win32 platform. To generate 64 bit configurations as well, you need to explicitly request them by defining the architectures to build for:

```
archs = x86 x86_64;
```

The name of 64 bit architecture is `x86_64` and not `x64` which is usually used under Windows because the architectures are also used under other platforms, notably OS X where universal binaries containing both 32 bit and 64 bit binaries would be built with the above value of `archs`.

2.4 Advanced Stuff

2.4.1 Generated Source Files

Bakefile supports custom compilation steps; this can be used both for files generated with some script and for compilation of unsupported file types.

Compiling a custom file is as simple as setting the `compile-commands` property on it to the command (or several commands) to compile the file, `outputs` property with the list of created files and optionally filling in additional dependencies:

```
toolsets = gnu vs2010;
program hello {
    sources { hello.cpp mygen.cpp mygen.desc }

    mygen.desc::compile-commands = "tools/generator.py -o %(out) %(in)";
```

(continues on next page)

(continued from previous page)

```
    mygen.desc::outputs = mygen.cpp;
    // add dependency on the generator script:
    mygen.desc::dependencies = tools/generator.py;
}
```

Notice that the generated files listed in `outputs` must be included in `sources` or `headers` section as well.

Additionally, any number of other dependency files can be added to the `dependencies` list. The command uses two placeholders, `%(in)` and `%(out)`, that are replaced with the name of the source file (`mygen.desc` in our example) and `outputs` respectively; both placeholders are optional. If there are multiple output files, `%(out0)`, `%(out1)`, ... placeholders can be used to access individual items in the list.

Perhaps a better would be to demonstrate how to use this to generate a grammar parser with Bison:

```
sources {
    main.cpp
    parser.ypp                  // Bison grammar file
    parser.cpp parser.hpp       // generated C++ parser
}

parser.ypp::compile-commands = "bison -o %(out0) %(in)"
parser.ypp::outputs = parser.cpp parser.hpp;
```

CHAPTER 3

Language reference

3.1 Statements, blocks, literals etc.

Statements in Bakefile are separated by semicolon (`;`) and code blocks are marked up in with `{` and `}`, as in C. See an example:

```
toolsets = gnu vs2010;
program hello {
    sources { hello.cpp }
}
```

In particular, expressions may span multiple lines without the need to escape newlines or enclose the expression in parenthesis:

```
os_files = foo.cpp
           bar.cpp
           ;
```

3.2 Values, types and literals

Similarly to the `make` syntax, quotes around literals are optional – anything not a keyword or special character or otherwise specially marked is a literal; specifically, a string literal.

Quoting is only needed when the literal contains whitespace or special characters such as = or quotes. Quoted strings are enclosed between `"` (double quote) or `'` (single quote) characters and may contain any characters except for the quotes. Additionally, backslash (`\`) can be used inside quoted strings to escape any character.[1]

The two kinds of quoting differ:

[1] A string literal containing quotes can therefore be written as, say, `"VERSION=\"1.2\""`; backslashes must be escaped as double backslashes (`"\\"`).

1. Double-quoted strings are interpolated. That is, variable references using `$(...)` (see below) are recognized and evaluated. If you want to use `$` as a literal, you must escape it (`\$`).
2. Single-quoted strings are literal, `$` doesn't have any special meaning and is treated as any other character.

Values in Bakefile are typed: properties have types associated with them and only values that are valid for that type can be assigned to them. The language isn't *strongly*-typed, though: conversions are performed whenever needed and possible, variables are untyped by default. Type checking primarily shows up when validating values assigned to properties.

The basic types are:

1. *Boolean* properties can be assigned the result of a boolean expression or one of the `true` or `false` literals.
2. *Strings*. Enough said.
3. *Lists* are items delimited with whitespace. Lists are typed and the items must all be of the same type. In the reference documentation, list types are described as "list of string", "list of path" etc.
4. *Paths* are file or directory paths and are described in more detail in the next section.
5. *IDs* are identifiers of targets.
6. *Enums* are used for properties where only a few possible values exist; the property cannot be set to anything other than one of the listed strings.
7. `AnyType` is the pseudo-type used for untyped variables or expressions with undetermined type.

3.3 Paths

File paths is a type that deserves more explanation. They are arguably the most important element in makefiles and project files both and any incorrectness in them would cause breakage.

All paths in bakefiles must be written using a notation similar to the Unix one, using `/` as the separator, and are always relative. By default, if you don't say otherwise and write the path as a normal Unix path (e.g. `src/main.cpp`), it's relative to the *source directory* (or *srcdir* for short). *Srcdir* is the implicitly assumed directory for the input files specified using relative paths. By default, it is the directory containing the bakefile itself but it can be changed as described below. Note that this may be – and often is – different from the location where the generated *output* files are written to.

This is usually the most convenient choice, but it's sometimes not sufficient. For such situations, Bakefile has the ability to *anchor* paths under a different root. This is done by adding a prefix of the form of `@<anchor>/` in front of the path. The following anchors are recognized:

1. `@srcdir`, as described above.
2. `@top_srcdir` is the top level source directory, i.e. *srcdir* of the top-most bakefile of the project. This is only different from @srcdir if this bakefile was included from another one as a submodule.
3. `@builddir` is the directory where build files of the current target are placed. Note that this is not where the generated makefiles or projects go either. It's often a dedicated directory just for the build artifacts and typically depends on make-time configuration. Visual Studio, for example, puts build files into `Debug/` and `Release/` subdirectories depending on the configuration selected. `@builddir` points to these directories.

Here are some examples showing common uses for the anchors:

```
sources {
    hello.cpp;  // relative to srcdir
    @builddir/generated_file.c;
```

(continues on next page)

(continued from previous page)

```
}
includedirs += @top_srcdir/include;
```

3.3.1 Changing *srcdir*

As mentioned above, `@srcdir` can be changed if its default value is inconvenient, as, for example, is the case when the bakefile itself is in a subdirectory of the source tree.

Take this for an example:

```
// build/bakefiles/foo.bkl
library foo {
    includedirs += ../../include;
    sources {
        ../../src/foo.cpp
        ../../src/bar.cpp
    }
}
```

This can be made much nicer using `srcdir`:

```
// build/bakefiles/foo.bkl
srcdir ../..;

library foo {
    includedirs += include;
    sources {
        src/foo.cpp
        src/bar.cpp
    }
}
```

The `srcdir` statement takes one argument, path to the new *srcdir* (relative to the location of the bakefile). It affects all `@srcdir`-anchored paths, including implicitly anchored ones, i.e. those without any explicit anchor, in the module (but not its submodules). Notably, (default) paths for generated files are also affected, because these too are relative to `@srcdir`.

Notice that because it affects the interpretation of all path expressions in the file, it can only be used before any assignments, target definitions etc. The only thing that can precede it is `requires`.

3.4 Variables and properties

Bakefile allows you to set arbitrary variables on any part of the model. Additionally, there are *properties*, which are pre-defined variables with a set meaning. Syntactically, there's no difference between the two. There's semantical difference in that the properties are usually typed and only values compatible with their type can be assigned to them. For example, you cannot assign arbitrary string to a *path* property or overwrite a read-only property.

3.4.1 Setting variables

Variables don't need to be declared; they are defined on first assignment. Assignment to variables is done in the usual way:

```
variable = value;
// Lists can be appended to, too:
main_sources = foo.cpp;
main_sources += bar.cpp third.cpp;
```

Occasionally, it is useful to set variables on other objects, not just in the current scope. For example, you may want to set per-file compilation flags, add custom build step for a particular source file or even modify a global variable. Bakefile uses operator *::* for this purpose, with semantics reminiscent of C++: any number of scopes delimited by *::* may precede the variable name, with leading *::* indicating global (i.e. current module) scope. Here's a simple example:

3.4.2 Referencing variables

Because literals aren't quoted, variables are referenced using the make-like `$(<varname>)` syntax:

```
platform = windows;
sources { os/$(platform).cpp }
```

A shorthand form, where the brackets are omitted, is also allowed when such use is unambiguous:[2]

```
if ( $toolset == gnu ) { ... }
```

Note that the substitution isn't done immediately. Instead, the reference is included in the object model of the bakefiles and is dereferenced at a later stage, when generating makefile and project files. Sometimes, they are kept in the generated files too.

This has two practical consequences:

1. It is possible to reference variables that are defined later in the bakefile without getting errors.
2. Definitions cannot be recursive, a variable must not reference itself. You cannot write this:

   ```
   defines = $(defines) SOME_MORE
   ```

   Use operator += instead:

   ```
   defines += SOME_MORE
   ```

3.5 Targets

Target definition consists of three things: the *type* of the target (an executable, a library etc.), its *ID* (the name, which usually corresponds to built file's name, but doesn't have to) and detailed specification of its properties:

```
type id {
    property = value;
    property = value;
    ...sources specification...
    ...more content...
}
```

(It's a bit more complicated than that, the content may contain conditional statements too, but that's the overall structure.)

[2] A typical example of *ambiguous* use is in a concatenation. You can't write `$toolset.cpp` because `.` is a valid part of a literal; it must be written as `$(toolset).cpp` so that it's clear which part is a variable name and which is a literal appended to the reference. For similar reasons, the shorthand form cannot be used in double-quoted strings.

3.5.1 Sources files

Source files are added to the target using the `sources` keyword, followed by the list of source files inside curly brackets. Note the sources list may contain any valid expression; in particular, references to variables are permitted.

It's possible to have multiple `sources` statements in the same target. Another use of `sources` appends the files to the list of sources, it doesn't overwrite it; the effect is the same as that of operator +=.

See an example:

```
program hello {
    sources {
        hello.cpp
        utils.cpp
    }

    // add some more sources later:
    sources { $(EXTRA_SOURCES) }
}
```

3.5.2 Headers

Syntax for headers specification is identical to the one used for source files, except that the `headers` keyword is used instead. The difference between sources and headers is that the latter may be used outside of the target (e.g. a library installs headers that are then used by users of the library).

3.6 Templates

It is often useful to share common settings or even code among multiple targets. This can be handled, to some degree, by setting properties such as `includedirs` globally, but more flexibility is often needed.

Bakefile provides a convenient way of doing just that: *templates*. A template is a named block of code that is applied and evaluated before target's own body. In a way, it's similar to C++ inheritance: targets correspond to derived classes and templates would be abstract base classes in this analogy.

Templates can be derived from another template; both targets and templates can be based on more than one template. They are applied in the order they are specified in, with base templates first and derived ones after them. Each template in the inheritance chain is applied exactly once, i.e. if a target uses the same template two or more times, its successive appearances are simply ignored.

Templates may contain any code that is valid inside target definition and may reference any variables defined in the target.

The syntax is similar to C++ inheritance syntax:

```
template common_stuff {
    defines += BUILDING;
}

template with_logging : common_stuff {
    defines += "LOGGING_ID=\"$(id)\"";
    libs += logging;
}

program hello : with_logging {
```

(continues on next page)

(continued from previous page)

```
    sources {
        hello.cpp
    }
}
```

Or equivalently:

```
template common_stuff {
    defines += BUILDING;
}

template with_logging {
    defines += "LOGGING_ID=\"$(id)\"";
    libs += logging;
}

program hello : common_stuff, with_logging {
    sources {
        hello.cpp
    }
}
```

3.7 Conditional statements

Any part of a bakefile may be enclosed in a conditional `if` statement. The syntax is similar to C/C++'s one:

```
defines = BUILD;
if ( $(toolset) == gnu )
    defines += LINUX;
```

In this example, the `defines` list will contain two items, `[BUILD, LINUX]` when generating makefiles for the `gnu` toolset and only one item, `BUILD`, for other toolsets. The condition doesn't have to be constant, it may reference e.g. options, where the value isn't known until make-time; Bakefile will correctly translate them into generated code.[3]

A long form with curly brackets is accepted as well; unlike the short form, this one can contain more than one statement:

```
if ( $(toolset) == gnu ) {
    defines += LINUX;
    sources { os/linux.cpp }
}
```

Conditional statements may be nested, too:

```
if ( $(build_tests) ) {
    program test {
        sources { main.cpp }
        if ( $(toolset) == gnu ) {
            defines += LINUX;
            sources { os/linux.cpp }
        }
```

(continues on next page)

[3] Although the syntax imposes few limits, it's not always possible to generate makefiles or projects with complicated conditional content even though the syntax supports it. In that case, Bakefile will exit with an explanatory error message.

(continued from previous page)

```
    }
}
```

The expression that specifies the condition uses C-style boolean operators: `&&` for *and*, `||` for *or*, `!` for *not* and `==` and `!=` for equality and inequality tests respectively.

3.8 Build configurations

A feature common to many IDEs is support for different build configurations, i.e. for building the same project using different compilation options. Bakefile generates the two standard "Debug" and "Release" configurations by default for the toolsets that usually use them (currently "vs*") and also supports the use of configurations with the makefile-based toolsets by allowing to specify `config=NameOfConfig` on make command line, e.g.

```
$ make config=Debug
# ... files are compiled with "-g" option and without optimizations ...
```

Notice that configuration names shouldn't be case-sensitive as `config=debug` is handled in the same way as `config=Debug` in make-based toolsets.

In addition to these two standard configurations, it is also possible to define your own *custom configurations*, which is especially useful for the project files which can't be customized as easily as the makefiles at build time.

Here is a step by step guide to doing this. First, you need to define the new configuration. This is done by using a configuration declaration in the global scope, i.e. outside of any target, e.g.:

```
configuration ExtraDebug : Debug {
}
```

The syntax for configuration definition is reminiscent of C++ class definition and, as could be expected, the identifier after the colon is the name of the *base configuration*. The new configuration inherits the variables defined in its base configuration.

Notice that all custom configurations must derive from another existing one, which can be either a standard "Debug" or "Release" configuration or a previously defined another custom configuration.

Defining a configuration doesn't do anything on its own, it also needs to be used by at least some targets. To do it, the custom configuration name must be listed in an assignment to the special `configurations` variable:

```
configurations = Debug ExtraDebug Release;
```

This statement can appear either in the global scope, like above, in which case it affects all the targets, or inside one or more targets, in which case the specified configuration is only used for these targets. So if you only wanted to enable extra debugging for "hello" executable you could do

```
program hello {
    configurations = Debug ExtraDebug Release;
}
```

However even if the configuration is present in the generated project files after doing all this, it is still not very useful as no custom options are defined for it. To change this, you will usually also want to set some project options conditionally depending on the configuration being used, e.g.:

```
program hello {
    if ( $(config) == ExtraDebug ) {
```

(continues on next page)

(continued from previous page)

```
        defines += EXTRA_DEBUG;
    }
}
```

`config` is a special variable automatically set by bakefile to the name of the current configuration and may be used in conditional expressions as any other variable.

For simple cases like the above, testing `config` explicitly is usually all you need but in more complex situations it might be preferable to define some variables inside the configuration definition and then test these variables instead. Here is a complete example doing the same thing as the above snippets using this approach:

```
configuration ExtraDebug : Debug {
    extra_debug = true;
}

configurations = Debug ExtraDebug Release;

program hello {
    if ( $(extra_debug) ) {
        defines += EXTRA_DEBUG;
    }
}
```

Note: As mentioned above, it is often unnecessary (although still possible) to define configurations for the makefile-based toolsets as it's always possible to just write `make CPPFLAGS=-DEXTRA_DEBUG` instead of using an "ExtraDebug" configuration from the example above with them. If you want to avoid such unnnecessary configurations in your makefiles, you could define them only conditionally, for example:

```
toolsets = gnu vs2010;
if ( $toolset == vs2010 && $config == ExtraDebug )
    defines += EXTRA_DEBUG;
```

would work as before in Visual Studio but would generate a simpler makefile.

3.9 Build settings

Sometimes, configurability provided by *configurations* is not enough and more flexible settings are required; e.g. configurable paths to 3rdparty libraries, tools and so on. Bakefile handles this with settings: variable-like constructs that are, unlike Bakefile variables, preserved in the generated output and can be modified by the user at make-time.

Settings are part of the object model and as such have a name and additional properties that affect their behavior. Defining a setting is similar to defining a target:

```
setting JDK_HOME {
    help = "Path to the JDK";
    default = /opt/jdk;
}
```

Notice that the setting object has some properties. You will almost always want to set the two shown in the above example. *help* is used to explain the setting to the user and *default* provides the default value to use if the user of the makefile doesn't specify anything else; both are optional. See *Setting properties* for the full list.

When you need to reference a setting, use the same syntax as when referencing variables:

```
includedirs += $(JDK_HOME)/include;
```

In fact, settings also act as variables defined at the highest (project) level. This means that they can be assigned to as well and some nice tricks are easily done:

```
setting LIBFOO_PATH {
    help = "Path to the Foo library";
    default = /opt/libfoo;
}

// On Windows, just use our own copy:
if ( $toolset == vs2010 )
    LIBFOO_PATH = @top_srcdir/3rdparty/libfoo;
```

This removes the user setting for toolsets that don't need it. Another handy use is to import some common code or use a submodule with configurable settings and just hard-code their values when you don't need the flexibility.

Note: Settings are currently only fully supported by makefiles, they are always replaced with their default values in the project files.

3.10 Submodules

A bakefile file – a *module* – can include other modules as its children. The `submodule` keyword is used for that:

```
submodule samples/hello/hello.bkl;
submodule samples/advanced/adv.bkl;
```

They are useful for organizing larger projects into more manageable chunks, similarly to how makefiles are used with recursive make. The submodules get their own makefiles (automatically invoked from the parent module's makefile) and a separate Visual Studio solution file is created for them by default as well. Typical uses include putting examples or tests into their own modules.

Submodules may only be included at the top level and cannot be included conditionally (i.e. inside an `if` statement).

3.11 Importing other files

There's one more way to organize source bakefiles in addition to submodules: direct import of another file's content. The syntax is similar to submodules one, using the `import` keyword:

```
// define variables, templates etc:
import common-defs.bkl;

program myapp { ... }
```

Import doesn't change the layout of output files, unlike `submodule`. Instead, it directly includes the content of the referenced file at the point of import. Think of it as a variation on C's `#include`.

Imports help with organizing large bakefiles into more manageable files. You could, for example, put commonly used variables or templates, files lists etc. into their own reusable files.

Notice that there are some important differences to `#include`:

1. A file is only imported once *in the current scope*, further imports are ignored. Specifically:
   (a) Second import of `foo.bkl` from the same module is ignored.
   (b) Import of `foo.bkl` from a submodule is ignored if it was already imported into its parent (or any of its ancestors).
   (c) If two sibling submodules both import `foo.bkl` and none of their ancestors does, then the file is imported into *both*. That's because their local scopes are independent of each other, so it isn't regarded as duplicate import.
2. An imported file may contain templates or configurations definitions and be included repeatedly (in the (1c) case above). This would normally result in errors, but Bakefile recognizes imported duplicates as identical and handles them gracefully.

The `import` keyword can only be included at the top level and cannot be done conditionally (i.e. inside an `if` statement).

3.12 Version checking

If a bakefile depends on features (or even syntax) not available in older versions, it is possible to declare this dependency using the `requires` keyword.

```
// Feature XYZ was added in Bakefile 1.1:
requires 1.1;
```

This statement causes fatal error if Bakefile version is older than the specified one.

3.13 Loading plugins

Standard Bakefile plugins are loaded automatically. But sometimes a custom plugin needed only for a specific project is needed and such plugins must be loaded explicitly, using the `plugin` keyword:

```
plugin my_compiler.py;
```

Its argument is a path to a valid Python file that will be loaded into the `bkl.plugins` module. You can also use full name of the module to make it clear the file is a Bakefile plugin:

```
plugin bkl.plugins.my_compiler.py;
```

See the *Writing Bakefile plugins* chapter for more information about plugins.

3.14 Comments

Bakefile uses C-style comments, in both the single-line and multi-line variants. Single-line comments look like this:

```
// we only generate code for GNU format for now
toolsets = gnu;
```

Multi-line comments can span several lines:

```
/*
   We only generate code for GNU format for now.
   This will change later, when we add Visual C++ support.
 */
toolsets = gnu;
```

They can also be included in an expression:

```
program hello {
    sources { hello.c /*main() impl*/ lib.c }
}
```

CHAPTER 4

Reference documentation

4.1 Global project properties

4.1.1 Properties

toolset (type: toolset)

The toolset makefiles or projects are being generated for. This property is set by Bakefile and can be used for performing toolset-specific tasks or modifications.

Read-only property

Allowed values: “gnu”, “gnu-osx”, “gnu-suncc”, “vs2003”, “vs2005”, “vs2008”, “vs2010”, “vs2012”, “vs2013”, “vs2015”, “vs2017”

Inheritable from parent: no

config (type: string)

Current configuration.

This property is set by Bakefile and can be used for performing per-configuration modifications. The value is one of the *configurations* values specified for the target.

See *Build configurations* for more information.

Read-only property

Inheritable from parent: no

arch (type: string)

Current architecture.

This property is set by Bakefile and can be used for performing per-architecture modifications (if the toolset supports it, which currently only Visual Studio does). The value is one of the *archs* values specified for the target.

Read-only property

Inheritable from parent: no

4.2 Module properties

4.2.1 Properties

toolsets (type: list of toolsets)

List of toolsets to generate makefiles/projects for.

Default: empty

Allowed values: “gnu”, “gnu-osx”, “gnu-suncc”, “vs2003”, “vs2005”, “vs2008”, “vs2010”, “vs2012”, “vs2013”, “vs2015”, “vs2017”

Inheritable from parent: yes

vs2008.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: *Visual Studio 2008 (vs2008)*

Default: same name as the module’s bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2008.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: *Visual Studio 2008 (vs2008)*

Default: True

Inheritable from parent: no

gnu.makefile (type: path)

Name of output file for module’s makefile.

Only for toolsets: *GNU toolchain for Unix systems (gnu)*

Default: GNUmakefile

Inheritable from parent: no

vs2005.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: same name as the module’s bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2005.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: True

Inheritable from parent: no

vs2017.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: Visual Studio 2017 (vs2017)

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2017.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: Visual Studio 2017 (vs2017)

Default: True

Inheritable from parent: no

gnu-suncc.makefile (type: path)

Name of output file for module's makefile.

Only for toolsets: GNU toolchain for Sun CC compiler (gnu-suncc)

Default: Makefile.suncc

Inheritable from parent: no

vs2015.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: Visual Studio 2015 (vs2015)

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2015.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: Visual Studio 2015 (vs2015)

Default: True

Inheritable from parent: no

vs2012.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: Visual Studio 2012 (vs2012)

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2012.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: Visual Studio 2012 (vs2012)

Default: True

Inheritable from parent: no

vs2013.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: Visual Studio 2013 (vs2013)

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2013.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: Visual Studio 2013 (vs2013)

Default: True

Inheritable from parent: no

vs2003.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: Visual Studio 2003 (vs2003)

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2003.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: Visual Studio 2003 (vs2003)

Default: True

Inheritable from parent: no

vs2010.solutionfile (type: path)

File name of the solution file for the module.

Only for toolsets: Visual Studio 2010 (vs2010)

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2010.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Only for toolsets: Visual Studio 2010 (vs2010)

Default: True

Inheritable from parent: no

gnu-osx.makefile (type: path)

Name of output file for module's makefile.

Only for toolsets: *GNU toolchain for OS X (gnu-osx)*

Default: Makefile.osx

Inheritable from parent: no

4.3 Setting properties

4.3.1 Properties

help (type: string)

Documentation for the setting. This will be used in the generated output to explain the setting to the user, if supported by the toolset.

Default: null

Inheritable from parent: no

default (type: any)

Default value of the setting, if any.

Default: null

Inheritable from parent: no

4.4 Toolsets

4.4.1 GNU toolchain for Unix systems *(gnu)*

GNU toolchain for Unix systems.

This toolset generates makefiles for the GNU toolchain – GNU Make, GCC compiler, GNU LD linker etc. – running on Unix system.

Currently, only Linux systems (or something sufficiently compatible) are supported. In particular, file extensions and linker behavior (symlinks, sonames) are assumed to be Linux ones.

See *GNU toolchain for OS X (gnu-osx)* for OS X variant.

Properties

Modules

gnu.makefile (type: path)

Name of output file for module's makefile.

Default: GNUmakefile

Inheritable from parent: no

4.4.2 GNU toolchain for OS X *(gnu-osx)*

GNU toolchain for OS X.

This toolset is for building on OS X using makefiles, not Xcode. It incorporates some of the oddities of OS X's toolchain and should be used instead of *GNU toolchain for Unix systems (gnu)*.

Properties

Modules

gnu-osx.makefile (type: path)

Name of output file for module's makefile.

Default: Makefile.osx

Inheritable from parent: no

4.4.3 GNU toolchain for Sun CC compiler *(gnu-suncc)*

GNU toolchain for Sun CC compiler.

This toolset is for building using the Sun CC (aka Oracle Studio) toolset.

Properties

Modules

gnu-suncc.makefile (type: path)

Name of output file for module's makefile.

Default: Makefile.suncc

Inheritable from parent: no

4.4.4 Visual Studio 2003 *(vs2003)*

Visual Studio 2003.

Special properties

This toolset supports the same special properties that *Visual Studio 2008 (vs2008)*. The only difference is that they are prefixed with `vs2003.option.` instead of `vs2008.option.`.

Properties

All targets

vs2003.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2003.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2003.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2003.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.4.5 Visual Studio 2005 *(vs2005)*

Visual Studio 2005.

Special properties

This toolset supports the same special properties that *Visual Studio 2008 (vs2008)*. The only difference is that they are prefixed with `vs2005.option.` instead of `vs2008.option.`.

Properties

All targets

vs2005.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2005.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2005.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2005.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.4.6 Visual Studio 2008 *(vs2008)*

Visual Studio 2008.

Special properties

In addition to the properties described below, it's possible to specify any of the `vcproj` properties directly in a bakefile. To do so, you have to set specially named variables on the target.

The variables are prefixed with `vs2008.option.`, followed by tool name and attribute name. For example:

- `vs2008.option.VCCLCompilerTool.EnableFunctionLevelLinking`
- `vs2008.option.VCLinkerTool.EnableCOMDATFolding`

Additionally, the following are supported for non-tool nodes:

- `vs2008.option.*` (attributes of the root `VisualStudioProject` node)
- `vs2008.option.Configuration.*` (`Configuration` node attributes)

These variables can be used in several places in bakefiles:

- In targets, to applied them as project's global settings.
- In modules, to apply them to all projects in the module and its submodules.
- On per-file basis, to modify file-specific settings.

Examples:

```
vs2008.option.VCCLCompilerTool.EnableFunctionLevelLinking = false;

crashrpt.cpp::vs2008.option.VCCLCompilerTool.ExceptionHandling = 2;
```

Properties

All targets

vs2008.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2008.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2008.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2008.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.4.7 Visual Studio 2010 *(vs2010)*

Visual Studio 2010.

Special properties

In addition to the properties described below, it's possible to specify any of the `vcxproj` properties directly in a bakefile. To do so, you have to set specially named variables on the target.

The variables are prefixed with `vs2010.option.`, followed by node name and property name. The following nodes are supported:

- `vs2010.option.Globals.*`
- `vs2010.option.Configuration.*`
- `vs2010.option.*` (this is the unnamed `PropertyGroup` with global settings such as `TargetName`)
- `vs2010.option.ClCompile.*`
- `vs2010.option.ResourceCompile.*`

- `vs2010.option.Link.*`
- `vs2010.option.Lib.*`
- `vs2010.option.Manifest.*`

These variables can be used in several places in bakefiles:

- In targets, to applied them as project's global settings.
- In modules, to apply them to all projects in the module and its submodules.
- On per-file basis, to modify file-specific settings.

Examples:

```
vs2010.option.GenerateManifest = false;
vs2010.option.Link.CreateHotPatchableImage = Enabled;

crashrpt.cpp::vs2010.option.ClCompile.ExceptionHandling = Async;
```

Properties

All targets

vs2010.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2010.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2010.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2010.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.4.8 Visual Studio 2012 *(vs2012)*

Visual Studio 2012.

Special properties

This toolset supports the same special properties that *Visual Studio 2010 (vs2010)*. The only difference is that they are prefixed with `vs2012.option.` instead of `vs2010.option.`, i.e. the nodes are:

- `vs2012.option.Globals.*`
- `vs2012.option.Configuration.*`
- `vs2012.option.*` (this is the unnamed `PropertyGroup` with global settings such as `TargetName`)
- `vs2012.option.ClCompile.*`
- `vs2012.option.ResourceCompile.*`
- `vs2012.option.Link.*`
- `vs2012.option.Lib.*`
- `vs2012.option.Manifest.*`

Properties

All targets

vs2012.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2012.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2012.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2012.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.4.9 Visual Studio 2013 *(vs2013)*

Visual Studio 2013.

Special properties

This toolset supports the same special properties that *Visual Studio 2010 (vs2010)*. The only difference is that they are prefixed with `vs2013.option.` instead of `vs2010.option.`, i.e. the nodes are:

- `vs2013.option.Globals.*`
- `vs2013.option.Configuration.*`
- `vs2013.option.*` (this is the unnamed `PropertyGroup` with global settings such as `TargetName`)
- `vs2013.option.ClCompile.*`
- `vs2013.option.ResourceCompile.*`
- `vs2013.option.Link.*`
- `vs2013.option.Lib.*`
- `vs2013.option.Manifest.*`

Properties

All targets

vs2013.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2013.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2013.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2013.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.4.10 Visual Studio 2015 *(vs2015)*

Visual Studio 2015.

Special properties

This toolset supports the same special properties that *Visual Studio 2010 (vs2010)*. The only difference is that they are prefixed with `vs2015.option.` instead of `vs2010.option.`, i.e. the nodes are:

- `vs2015.option.Globals.*`
- `vs2015.option.Configuration.*`
- `vs2015.option.*` (this is the unnamed `PropertyGroup` with global settings such as `TargetName`)
- `vs2015.option.ClCompile.*`
- `vs2015.option.ResourceCompile.*`
- `vs2015.option.Link.*`
- `vs2015.option.Lib.*`
- `vs2015.option.Manifest.*`

Properties

All targets

vs2015.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2015.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2015.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2015.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.4.11 Visual Studio 2017 *(vs2017)*

Visual Studio 2017.

Special properties

This toolset supports the same special properties that *Visual Studio 2010 (vs2010)*. The only difference is that they are prefixed with `vs2017.option.` instead of `vs2010.option.`, i.e. the nodes are:

- `vs2017.option.Globals.*`
- `vs2017.option.Configuration.*`
- `vs2017.option.*` (this is the unnamed `PropertyGroup` with global settings such as `TargetName`)
- `vs2017.option.ClCompile.*`
- `vs2017.option.ResourceCompile.*`
- `vs2017.option.Link.*`
- `vs2017.option.Lib.*`
- `vs2017.option.Manifest.*`

Properties

All targets

vs2017.projectfile (type: path)

File name of the project for the target.

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2017.guid (type: string)

GUID of the project.

Default: automatically generated

Inheritable from parent: no

Modules

vs2017.solutionfile (type: path)

File name of the solution file for the module.

Default: same name as the module's bakefile, with `.sln` extension, in `@srcdir`

Inheritable from parent: no

vs2017.generate-solution (type: bool)

Whether to generate solution file for the module. Set to `false` if you want to omit the solution, e.g. for some submodules with only a single target.

Default: True

Inheritable from parent: no

4.5 Targets

4.5.1 Custom action script *(action)*

Custom action script.

Action targets execute arbitrary commands. They can be used to do various tasks that don't fit the model of compiling or creating files, such as packaging files, installing, uploading, running tests and so on.

Actions are currently only supported by makefile-based toolsets. See the `pre-build-commands` and `post-build-commands` properties for another alternative that is supported by Visual Studio projects.

If the optional `outputs` property is specified, the action is supposed to generate the files listed in this property. This means that other targets depending on this action will depend on these files in the generated makefile, instead of depending on the phony target for an action without outputs and also that these files will be cleaned up by the `clean` target of the generated makefile.

```
action osx-bundle
{
  deps = test;
  commands = "mkdir -p Test.app/Contents/MacOS"
             "cp -f test Test.app/Contents/MacOS/test"
             ;
}
```

Properties

commands (type: list of strings)

List of commands to run.

Default: empty

Inheritable from parent: no

outputs (type: list of paths)

Output files created by this action, if any.

Default: null

Inheritable from parent: no

id (type: id)

Target's unique name (ID).

Read-only property

Inheritable from parent: no

deps (type: list of ids)

Dependencies of the target (list of IDs).

The dependencies are handled in target-specific ways. At the very least, they are added to the list of dependencies in generated makefiles or projects to ensure correct build order. Some targets may be smart about some kinds of the dependencies and do more.

In particular, compiled targets (executables, DLLs) will automatically link against all libraries found in *deps*.

Default: empty

Inheritable from parent: no

pre-build-commands (type: list of strings)

Custom commands to run before building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

post-build-commands (type: list of strings)

Custom commands to run after building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

configurations (type: list of strings)

List of configurations to use for this target.

See *Build configurations* for more information.

Default: Debug Release

Inheritable from parent: yes

vs2008.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2008.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2008 (vs2008)*

Default: automatically generated

Inheritable from parent: no

vs2005.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2005.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: automatically generated

Inheritable from parent: no

vs2017.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2017.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: automatically generated

Inheritable from parent: no

vs2015.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2015.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: automatically generated

Inheritable from parent: no

vs2012.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2012.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: automatically generated

Inheritable from parent: no

vs2013.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2013.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: automatically generated

Inheritable from parent: no

vs2003.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2003.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: automatically generated

Inheritable from parent: no

vs2010.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2010.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: automatically generated

Inheritable from parent: no

4.5.2 External build system *(external)*

External build system.

This target type is used to invoke makefiles or project files not implemented in Bakefile, for example to build 3rd party libraries.

Currently, only Visual Studio projects (vcproj, vcxproj, csproj) are supported and only when using a Visual Studio toolset.

Properties

file (type: path)

File name of the external makefile or project.

Required property

Inheritable from parent: no

id (type: id)

Target's unique name (ID).

Read-only property

Inheritable from parent: no

deps (type: list of ids)

Dependencies of the target (list of IDs).

The dependencies are handled in target-specific ways. At the very least, they are added to the list of dependencies in generated makefiles or projects to ensure correct build order. Some targets may be smart about some kinds of the dependencies and do more.

In particular, compiled targets (executables, DLLs) will automatically link against all libraries found in *deps*.

Default: empty

Inheritable from parent: no

pre-build-commands (type: list of strings)

Custom commands to run before building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

post-build-commands (type: list of strings)

Custom commands to run after building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

configurations (type: list of strings)

List of configurations to use for this target.

See *Build configurations* for more information.

Default: Debug Release

Inheritable from parent: yes

vs2008.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2008.guid (type: string)

GUID of the project.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: automatically generated

Inheritable from parent: no

vs2005.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2005 (vs2005)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2005.guid (type: string)

GUID of the project.

Only for toolsets: Visual Studio 2005 (vs2005)

Default: automatically generated

Inheritable from parent: no

vs2017.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2017 (vs2017)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2017.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: automatically generated

Inheritable from parent: no

vs2015.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2015.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: automatically generated

Inheritable from parent: no

vs2012.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2012.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: automatically generated

Inheritable from parent: no

vs2013.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2013.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: automatically generated

Inheritable from parent: no

vs2003.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2003.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: automatically generated

Inheritable from parent: no

vs2010.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2010.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: automatically generated

Inheritable from parent: no

4.5.3 Static library *(library)*

Static library.

Properties

basename (type: string)

Library name.

This is not full filename or even path, it's only its base part, to which platform-specific prefix (if applicable) and extension are added. By default, it's the same as target's ID, but it can be changed e.g. if the filename should contain version number, which would be impractical to use as target identifier in the bakefile.

```
library foo {
  // use e.g. libfoo24.a on Unix and foo24.lib
  basename = foo$(vermajor)$(verminor);
}
```

Default: $(id)

Inheritable from parent: no

sources (type: list of paths)

Source files.

Default: empty

Inheritable from parent: no

headers (type: list of paths)

Header files.

Default: empty

Inheritable from parent: no

defines (type: list of strings)

List of preprocessor macros to define.

Default: empty

Inheritable from parent: yes

includedirs (type: list of paths)

Directories where to look for header files.

Default: empty

Inheritable from parent: yes

warnings (type: warnings)

Warning level for the compiler.

Use `no` to completely disable warning, `minimal` to show only the most important warning messages, `all` to enable all warnings that usually don't result in false positives and `max` to enable absolutely all warnings.

Default: default

Allowed values: "no", "minimal", "default", "all", "max"

Inheritable from parent: yes

compiler-options (type: list of strings)

Additional compiler options common to all C-like compilers (C, C++, Objective-C, Objective-C++).

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

c-compiler-options (type: list of strings)

Additional options for C compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

cxx-compiler-options (type: list of strings)

Additional options for C++ compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

libs (type: list of strings)

Additional libraries to link with.

Do not use this property to link with libraries built as part of your project; use *deps* for that.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

libdirs (type: list of paths)

Additional directories where to look for libraries.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

link-options (type: list of strings)

Additional linker options.

Note that the options are compiler/linker-specific and so this property should only be set conditionally for particular compilers that recognize the options.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

archs (type: list of architectures)

Architectures to compile for.

Adds support for building binaries for specified architectures, if supported by the toolset. Support may take the form of either multi-arch binaries (OS X) or additional build configurations (Visual Studio).

The default empty value means to do whatever the default behavior of the toolset is.

Currently only supported on OS X and in Visual Studio.

Default: null

Allowed values: “x86”, “x86_64”

Inheritable from parent: yes

win32-crt-linkage (type: linkage)

How to link against the C Runtime Library.

If `dll` (the default), the executable may depend on some DLLs provided by the compiler. If `static` then a static version of the CRT is linked directly into the executable.

Default: dll

Allowed values: "static", "dll"

Inheritable from parent: yes

win32-unicode (type: bool)

Compile win32 code in Unicode mode? If enabled, `_UNICODE` symbol is defined and the wide character entry point (`WinMain`, ...) is used.

Default: True

Inheritable from parent: yes

outputdir (type: path)

Directory where final binaries are put.

Note that this is not the directory for intermediate files such as object files – these are put in `@builddir`. By default, output location is the same, `@builddir`, but can be overwritten to for example put all executables into `bin/` subdirectory.

Default: @builddir/

Inheritable from parent: yes

pic (type: bool)

Compile position-independent code.

By default, libraries (both shared and static, because the latter could be linked into a shared lib too) are linked with -fPIC and executables are not.

Default: None

Inheritable from parent: yes

multithreading (type: bool)

Enable support for multithreading.

MT support is enabled by default, but can be disabled when not needed.

On Unix, this option causes the use of pthreads library. Visual Studio always uses MT-safe CRT, even if this setting is disabled.

Default: True

Inheritable from parent: yes

id (type: id)

Target's unique name (ID).

Read-only property

Inheritable from parent: no

deps (type: list of ids)

Dependencies of the target (list of IDs).

The dependencies are handled in target-specific ways. At the very least, they are added to the list of dependencies in generated makefiles or projects to ensure correct build order. Some targets may be smart about some kinds of the dependencies and do more.

In particular, compiled targets (executables, DLLs) will automatically link against all libraries found in *deps*.

Default: empty

Inheritable from parent: no

pre-build-commands (type: list of strings)

Custom commands to run before building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

post-build-commands (type: list of strings)

Custom commands to run after building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

configurations (type: list of strings)

List of configurations to use for this target.

See *Build configurations* for more information.

Default: Debug Release

Inheritable from parent: yes

vs2008.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2008.guid (type: string)

GUID of the project.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: automatically generated

Inheritable from parent: no

vs2005.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2005.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: automatically generated

Inheritable from parent: no

vs2017.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2017.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: automatically generated

Inheritable from parent: no

vs2015.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2015.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: automatically generated

Inheritable from parent: no

vs2012.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2012.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: automatically generated

Inheritable from parent: no

vs2013.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2013.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: automatically generated

Inheritable from parent: no

vs2003.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2003.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: automatically generated

Inheritable from parent: no

vs2010.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2010.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: automatically generated

Inheritable from parent: no

4.5.4 Runtime-loaded dynamic module (plugin) *(loadable-module)*

Runtime-loaded dynamic module (plugin).

Properties

basename (type: string)

Base name of the loadable module.

This is not full filename or even path, it's only its base part, to which platform-specific prefix and/or extension are added. By default, it's the same as target's ID, but it can be changed e.g. if the filename should contain version number, which would be impractical to use as target identifier in the bakefile.

```
loadable-module myplugin {
  basename = myplugin-v1;
}
```

Default: $(id)

Inheritable from parent: no

extension (type: string)

File extension of the module, including the leading dot.

By default, native extension for shared libraries (e.g. ".dll" on Windows) is used.

```
loadable-module excel_plugin {
  extension = .xll;
}
```

Default: null

Inheritable from parent: no

sources (type: list of paths)

Source files.

Default: empty

Inheritable from parent: no

headers (type: list of paths)

Header files.

Default: empty

Inheritable from parent: no

defines (type: list of strings)

List of preprocessor macros to define.

Default: empty

Inheritable from parent: yes

includedirs (type: list of paths)

Directories where to look for header files.

Default: empty

Inheritable from parent: yes

warnings (type: warnings)

Warning level for the compiler.

Use `no` to completely disable warning, `minimal` to show only the most important warning messages, `all` to enable all warnings that usually don't result in false positives and `max` to enable absolutely all warnings.

Default: default

Allowed values: "no", "minimal", "default", "all", "max"

Inheritable from parent: yes

compiler-options (type: list of strings)

Additional compiler options common to all C-like compilers (C, C++, Objective-C, Objective-C++).

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

c-compiler-options (type: list of strings)

Additional options for C compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

cxx-compiler-options (type: list of strings)

Additional options for C++ compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

libs (type: list of strings)

Additional libraries to link with.

Do not use this property to link with libraries built as part of your project; use *deps* for that.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

libdirs (type: list of paths)

Additional directories where to look for libraries.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

link-options (type: list of strings)

Additional linker options.

Note that the options are compiler/linker-specific and so this property should only be set conditionally for particular compilers that recognize the options.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

archs (type: list of architectures)

Architectures to compile for.

Adds support for building binaries for specified architectures, if supported by the toolset. Support may take the form of either multi-arch binaries (OS X) or additional build configurations (Visual Studio).

The default empty value means to do whatever the default behavior of the toolset is.

Currently only supported on OS X and in Visual Studio.

Default: null

Allowed values: "x86", "x86_64"

Inheritable from parent: yes

win32-crt-linkage (type: linkage)

How to link against the C Runtime Library.

If `dll` (the default), the executable may depend on some DLLs provided by the compiler. If `static` then a static version of the CRT is linked directly into the executable.

Default: dll

Allowed values: "static", "dll"

Inheritable from parent: yes

win32-unicode (type: bool)

Compile win32 code in Unicode mode? If enabled, `_UNICODE` symbol is defined and the wide character entry point (`WinMain`, ...) is used.

Default: True

Inheritable from parent: yes

outputdir (type: path)

Directory where final binaries are put.

Note that this is not the directory for intermediate files such as object files – these are put in `@builddir`. By default, output location is the same, `@builddir`, but can be overwritten to for example put all executables into `bin/` subdirectory.

Default: @builddir/

Inheritable from parent: yes

pic (type: bool)

Compile position-independent code.

By default, libraries (both shared and static, because the latter could be linked into a shared lib too) are linked with -fPIC and executables are not.

Default: None

Inheritable from parent: yes

multithreading (type: bool)

Enable support for multithreading.

MT support is enabled by default, but can be disabled when not needed.

On Unix, this option causes the use of pthreads library. Visual Studio always uses MT-safe CRT, even if this setting is disabled.

Default: True

Inheritable from parent: yes

id (type: id)

Target's unique name (ID).

Read-only property

Inheritable from parent: no

deps (type: list of ids)

Dependencies of the target (list of IDs).

The dependencies are handled in target-specific ways. At the very least, they are added to the list of dependencies in generated makefiles or projects to ensure correct build order. Some targets may be smart about some kinds of the dependencies and do more.

In particular, compiled targets (executables, DLLs) will automatically link against all libraries found in *deps*.

Default: empty

Inheritable from parent: no

pre-build-commands (type: list of strings)

Custom commands to run before building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

post-build-commands (type: list of strings)

Custom commands to run after building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

configurations (type: list of strings)

List of configurations to use for this target.

See *Build configurations* for more information.

Default: Debug Release

Inheritable from parent: yes

vs2008.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2008.guid (type: string)

GUID of the project.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: automatically generated

Inheritable from parent: no

vs2005.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2005 (vs2005)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2005.guid (type: string)

GUID of the project.

Only for toolsets: Visual Studio 2005 (vs2005)

Default: automatically generated

Inheritable from parent: no

vs2017.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2017 (vs2017)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2017.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: automatically generated

Inheritable from parent: no

vs2015.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2015.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: automatically generated

Inheritable from parent: no

vs2012.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2012.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: automatically generated

Inheritable from parent: no

vs2013.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2013.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: automatically generated

Inheritable from parent: no

vs2003.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2003.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: automatically generated

Inheritable from parent: no

vs2010.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2010.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: automatically generated

Inheritable from parent: no

4.5.5 Executable program *(program)*

Executable program.

Properties

basename (type: string)

Base name of the executable.

This is not full filename or even path, it's only its base part, to which platform-specific extension is added. By default, it's the same as target's ID, but it can be changed e.g. if the filename should contain version number, which would be impractical to use as target identifier in the bakefile.

```
program mytool {
  // use mytool2.exe or /usr/bin/mytool2
  basename = $(id)$(vermajor);
}
```

Default: $(id)

Inheritable from parent: no

win32-subsystem (type: subsystem)

Windows subsystem the executable runs in. Must be set to `windows` for console-less applications.

Default: console

Allowed values: "console", "windows"

Inheritable from parent: yes

sources (type: list of paths)

Source files.

Default: empty

Inheritable from parent: no

headers (type: list of paths)

Header files.

Default: empty

Inheritable from parent: no

defines (type: list of strings)

List of preprocessor macros to define.

Default: empty

Inheritable from parent: yes

includedirs (type: list of paths)

Directories where to look for header files.

Default: empty

Inheritable from parent: yes

warnings (type: warnings)

Warning level for the compiler.

Use `no` to completely disable warning, `minimal` to show only the most important warning messages, `all` to enable all warnings that usually don't result in false positives and `max` to enable absolutely all warnings.

Default: default

Allowed values: "no", "minimal", "default", "all", "max"

Inheritable from parent: yes

compiler-options (type: list of strings)

Additional compiler options common to all C-like compilers (C, C++, Objective-C, Objective-C++).

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

c-compiler-options (type: list of strings)

Additional options for C compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

cxx-compiler-options (type: list of strings)

Additional options for C++ compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

libs (type: list of strings)

Additional libraries to link with.

Do not use this property to link with libraries built as part of your project; use *deps* for that.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

libdirs (type: list of paths)

Additional directories where to look for libraries.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

link-options (type: list of strings)

Additional linker options.

Note that the options are compiler/linker-specific and so this property should only be set conditionally for particular compilers that recognize the options.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

archs (type: list of architectures)

Architectures to compile for.

Adds support for building binaries for specified architectures, if supported by the toolset. Support may take the form of either multi-arch binaries (OS X) or additional build configurations (Visual Studio).

The default empty value means to do whatever the default behavior of the toolset is.

Currently only supported on OS X and in Visual Studio.

Default: null

Allowed values: “x86”, “x86_64”

Inheritable from parent: yes

win32-crt-linkage (type: linkage)

How to link against the C Runtime Library.

If `dll` (the default), the executable may depend on some DLLs provided by the compiler. If `static` then a static version of the CRT is linked directly into the executable.

Default: dll

Allowed values: “static”, “dll”

Inheritable from parent: yes

win32-unicode (type: bool)

Compile win32 code in Unicode mode? If enabled, `_UNICODE` symbol is defined and the wide character entry point (`WinMain`, ...) is used.

Default: True

Inheritable from parent: yes

outputdir (type: path)

Directory where final binaries are put.

Note that this is not the directory for intermediate files such as object files – these are put in `@builddir`. By default, output location is the same, `@builddir`, but can be overwritten to for example put all executables into `bin/` subdirectory.

Default: @builddir/

Inheritable from parent: yes

pic (type: bool)

Compile position-independent code.

By default, libraries (both shared and static, because the latter could be linked into a shared lib too) are linked with -fPIC and executables are not.

Default: None

Inheritable from parent: yes

multithreading (type: bool)

Enable support for multithreading.

MT support is enabled by default, but can be disabled when not needed.

On Unix, this option causes the use of pthreads library. Visual Studio always uses MT-safe CRT, even if this setting is disabled.

Default: True

Inheritable from parent: yes

id (type: id)

Target’s unique name (ID).

Read-only property

Inheritable from parent: no

deps (type: list of ids)

Dependencies of the target (list of IDs).

The dependencies are handled in target-specific ways. At the very least, they are added to the list of dependencies in generated makefiles or projects to ensure correct build order. Some targets may be smart about some kinds of the dependencies and do more.

In particular, compiled targets (executables, DLLs) will automatically link against all libraries found in *deps*.

Default: empty

Inheritable from parent: no

pre-build-commands (type: list of strings)

Custom commands to run before building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

post-build-commands (type: list of strings)

Custom commands to run after building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

configurations (type: list of strings)

List of configurations to use for this target.

See *Build configurations* for more information.

Default: Debug Release

Inheritable from parent: yes

vs2008.projectfile (type: path)

File name of the project for the target.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2008.guid (type: string)

GUID of the project.

Only for toolsets: Visual Studio 2008 (vs2008)

Default: automatically generated

Inheritable from parent: no

vs2005.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2005.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: automatically generated

Inheritable from parent: no

vs2017.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2017.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: automatically generated

Inheritable from parent: no

vs2015.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2015.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: automatically generated

Inheritable from parent: no

vs2012.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2012.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: automatically generated

Inheritable from parent: no

vs2013.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2013.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: automatically generated

Inheritable from parent: no

vs2003.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2003.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: automatically generated

Inheritable from parent: no

vs2010.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2010.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: automatically generated

Inheritable from parent: no

4.5.6 Dynamically loaded library *(shared-library)*

Dynamically loaded library.

Properties

basename (type: string)

Base name of the library.

This is not full filename or even path, it's only its base part, to which platform-specific prefix and/or extension are added. By default, it's the same as target's ID, but it can be changed e.g. if the filename should contain version number, which would be impractical to use as target identifier in the bakefile.

```
shared-library utils {
  // use myapp_utils.lib, myapp_utils.dll, libmyapp_utils.so
  basename = myapp_utils;
}
```

Default: $(id)

Inheritable from parent: no

sources (type: list of paths)

Source files.

Default: empty

Inheritable from parent: no

headers (type: list of paths)

Header files.

Default: empty

Inheritable from parent: no

defines (type: list of strings)

List of preprocessor macros to define.

Default: empty

Inheritable from parent: yes

includedirs (type: list of paths)

Directories where to look for header files.

Default: empty

Inheritable from parent: yes

warnings (type: warnings)

Warning level for the compiler.

Use `no` to completely disable warning, `minimal` to show only the most important warning messages, `all` to enable all warnings that usually don't result in false positives and `max` to enable absolutely all warnings.

Default: default

Allowed values: “no”, “minimal”, “default”, “all”, “max”

Inheritable from parent: yes

compiler-options (type: list of strings)

Additional compiler options common to all C-like compilers (C, C++, Objective-C, Objective-C++).

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

c-compiler-options (type: list of strings)

Additional options for C compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

cxx-compiler-options (type: list of strings)

Additional options for C++ compiler.

Note that the options are compiler-specific and so this property should only be set conditionally for particular compilers that recognize the options.

Default: empty

Inheritable from parent: yes

libs (type: list of strings)

Additional libraries to link with.

Do not use this property to link with libraries built as part of your project; use *deps* for that.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

libdirs (type: list of paths)

Additional directories where to look for libraries.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

link-options (type: list of strings)

Additional linker options.

Note that the options are compiler/linker-specific and so this property should only be set conditionally for particular compilers that recognize the options.

When this list is non-empty on a *Static library (library)*, it will be used when linking executables that use the library.

Default: empty

Inheritable from parent: yes

archs (type: list of architectures)

Architectures to compile for.

Adds support for building binaries for specified architectures, if supported by the toolset. Support may take the form of either multi-arch binaries (OS X) or additional build configurations (Visual Studio).

The default empty value means to do whatever the default behavior of the toolset is.

Currently only supported on OS X and in Visual Studio.

Default: null

Allowed values: "x86", "x86_64"

Inheritable from parent: yes

win32-crt-linkage (type: linkage)

How to link against the C Runtime Library.

If `dll` (the default), the executable may depend on some DLLs provided by the compiler. If `static` then a static version of the CRT is linked directly into the executable.

Default: dll

Allowed values: "static", "dll"

Inheritable from parent: yes

win32-unicode (type: bool)

Compile win32 code in Unicode mode? If enabled, `_UNICODE` symbol is defined and the wide character entry point (`WinMain`, ...) is used.

Default: True

Inheritable from parent: yes

outputdir (type: path)

Directory where final binaries are put.

Note that this is not the directory for intermediate files such as object files – these are put in `@builddir`. By default, output location is the same, `@builddir`, but can be overwritten to for example put all executables into `bin/` subdirectory.

Default: @builddir/

Inheritable from parent: yes

pic (type: bool)

Compile position-independent code.

By default, libraries (both shared and static, because the latter could be linked into a shared lib too) are linked with -fPIC and executables are not.

Default: None

Inheritable from parent: yes

multithreading (type: bool)

Enable support for multithreading.

MT support is enabled by default, but can be disabled when not needed.

On Unix, this option causes the use of pthreads library. Visual Studio always uses MT-safe CRT, even if this setting is disabled.

Default: True

Inheritable from parent: yes

id (type: id)

Target's unique name (ID).

Read-only property

Inheritable from parent: no

deps (type: list of ids)

Dependencies of the target (list of IDs).

The dependencies are handled in target-specific ways. At the very least, they are added to the list of dependencies in generated makefiles or projects to ensure correct build order. Some targets may be smart about some kinds of the dependencies and do more.

In particular, compiled targets (executables, DLLs) will automatically link against all libraries found in *deps*.

Default: empty

Inheritable from parent: no

pre-build-commands (type: list of strings)

Custom commands to run before building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

post-build-commands (type: list of strings)

Custom commands to run after building the target.

The value is a list of shell commands to run. Notice that the commands are platform-specific and so typically need to be set conditionally depending on the value of `toolset`.

Currently only implemented by Visual Studio.

Default: empty

Inheritable from parent: no

configurations (type: list of strings)

List of configurations to use for this target.

See *Build configurations* for more information.

Default: Debug Release

Inheritable from parent: yes

vs2008.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2008 (vs2008)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2008.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2008 (vs2008)*

Default: automatically generated

Inheritable from parent: no

vs2005.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2005.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2005 (vs2005)*

Default: automatically generated

Inheritable from parent: no

vs2017.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2017.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2017 (vs2017)*

Default: automatically generated

Inheritable from parent: no

vs2015.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2015.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2015 (vs2015)*

Default: automatically generated

Inheritable from parent: no

vs2012.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2012.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2012 (vs2012)*

Default: automatically generated

Inheritable from parent: no

vs2013.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2013.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2013 (vs2013)*

Default: automatically generated

Inheritable from parent: no

vs2003.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2003.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2003 (vs2003)*

Default: automatically generated

Inheritable from parent: no

vs2010.projectfile (type: path)

File name of the project for the target.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: `$(id).vcxproj` in the same directory as the `.sln` file

Inheritable from parent: no

vs2010.guid (type: string)

GUID of the project.

Only for toolsets: *Visual Studio 2010 (vs2010)*

Default: automatically generated

Inheritable from parent: no

CHAPTER 5

Extending and developing Bakefile

5.1 Writing Bakefile plugins

As mentioned in *Loading plugins*, it is possible to use plugins written in Python to extend Bakefile functionality. The most common types of plugins are those defining custom tool sets, i.e. new output formats, or custom build steps, allowing to perform more or less arbitrary actions.

But the simplest possible custom step plugin may actually not do anything at all, but just define some properties that can be used in bakefiles. For example, here is a complete example of a plugin:

```
from bkl.api import CustomStep, Property
from bkl.vartypes import StringType

from datetime import date

class MyVersion(CustomStep):
    """
    Simple bakefile plugin defining MY_VERSION property.

    The value of the version is just the current date.
    """
    name = "my_version"

    properties_project = [
        Property("MY_VERSION",
                 type=StringType(),
                 default=data.today().isoformat(),
                 inheritable=False,
                 readonly=True),
    ]
```

This plugin can then be used in the following way:

```
plugin bkl.plugins.my_version.py;

program my_program {
    basename = my_program-$(MY_VERSION);
    ...
}
```

Of course, a more realistic example would use something other than just the date of the last Bakefile execution as version. As plugin is just a normal Python script, there are a lot of possibilities, for example it could extract the version from the VCS used by the program or read it from some file inside the source tree.

5.2 Reference

5.2.1 API reference documentation

bkl.model – project model

class bkl.model.**Configuration**(*name*, *base*, *is_debug*, *source_pos=None*)

Bases: object

Class representing a configuration.

Each model has two special configurations, Debug and Release, predefined.

name
Name of the configuration

base
Base configuration this one is derived from, as a *Configuration* instance, or None for "Debug" and "Release" configurations.

is_debug
Is this a debug configuration?

source_pos
Source code position of object's definition, or None.

create_derived(*new_name*, *source_pos=None*)
Returns a new copy of this configuration with a new name.

derived_from(*cfg*)
Returns true if *self* derives, directly or indirectly, from configuration *cfg*.

Returns 0 if not derived. Returns degree of inheritance if derived: 1 if *cfg* is a direct base, 2 if it is a base of *self*'s base etc.

class bkl.model.**ConfigurationProxy**(*model*, *config*)

Bases: object

Proxy for accessing model part's variables as configuration-specific. All expressions obtained using operator[] are processed to remove conditionals depending on the value of "config", by substituting appropriate value according to the configuration name passed to proxy's constructor.

See bkl.model.ModelPartWithConfigurations.configurations() for more information.

apply_subst(*value*)
Applies the proxy's magic on given value. This is useful for when the proxy cannot be simply used in

place of a real target. For example, NativeLinkedType.get_ldlibs() inspects other targets too and so the proxy is only partially effective.

class bkl.model.**ConfigurationsPropertyMixin**
Mixin class for implementation configurations property.

configurations
Returns all configurations for this model part (e.g. a target), as ConfigurationProxy objects that can be used similarly to the way the model part object itself is. In particular, it's [] operator works the same.

The proxies are intended to be used in place of targets in code that needs to get per-configuration values of properties.

```
>>> for cfg in target.configurations:
...     outdir = cfg["outdir"]
```

class bkl.model.**ModelPart**(*parent*, *source_pos=None*)
Bases: object

Base class for model "parts", i.e. projects, modules or targets. Basically, anything that can have variables on it.

name
Name of the part – e.g. target name, filename of source file etc.

variables
Dictionary of all variables defined in global scope in this module

parent
Parent model part of this one, i.e. the part one step higher in object model hierarchy (e.g. module for a target). May only be None for toplevel part (the project). Dictionary of all variables defined in global scope in this module

source_pos
Source code position of object's definition, or None.

add_variable(*var*)
Adds a new variable object.

all_variables()
Returns iterator over all variables in the target. Works recursively, i.e. scans all modules and targets under this object too.

child_parts()
Yields model parts that are (direct) children of this.

enum_props()
Enumerates properties defined on this object.

Like *get_prop()*, this method doesn't work recursively upwards, but lists only properties that are defined for this scope.

See also:

get_prop()

get_child_part_by_name(*name*)
Returns child of this part of the model with given name (e.g. target with that name). Throws if not found.

get_matching_prop_with_inheritance(*name*)
Like *get_prop()*, but looks for inheritable properties defined for child scopes too (e.g. returns per-target property 'outputdir' even in module scope).

(This is used for assigning into variables and validating them against properties. Don't use unless you know what you're doing.)

get_prop(*name*)
Try to get a property *name*. Called by get_variable_value() if no variable with such name could be found.

Note that unlike *get_variable()*, this one doesn't work recursively upwards, but finds only properties that are defined for this scope.

See also:

enum_props()

get_variable(*name*)
Returns variable object for given variable or None if it is not defined *at this scope*.

This method does not do any recursive resolution or account for properties and their default values; it merely gets the variable object if it is defined at this scope.

See also:

get_variable_value(), *resolve_variable()*

get_variable_value(*name*)
Similar to get_variable(), but returns the expression with variable's value. Throws and exception if the variable isn't defined, neither in this scope or in any of its parent scopes.

If variable *name* is not explicitly defined, but a property with the same name exists in this scope, then its default value is used.

If the variable is not defined in this scope at all, looks in the parent – but only if *name* doesn't correspond to non-inheritable property. In other words, fails only if the variable isn't defined for use in this scope.

Throws if the value cannot be found.

As a shorthand syntax for this function, key indices may be used: >>> target["includedirs"]

See also:

resolve_variable()

is_variable_explicitly_set(*name*)
Returns true if the variable was set in the bakefiles explicitly.

is_variable_null(*name*)
Returns true if the variable is unset or null (which amounts to not being set in this toolset / under current condition).

make_variables_for_missing_props(*toolset*)
Creates variables for properties that don't have variables set yet.

Parameters **toolset** – Name of the toolset to generate for. Properties specific to other toolsets are ignored for efficiency.

resolve_variable(*name*)
Returns variable object for given variable or None if it is not defined.

Unlike *get_variable()*, this method does perform recursive resolution and finds the variable (if it exists) according to the same rules that apply to *$(...)* expressions and to *get_variable_value()*.

See also:

get_variable(), *get_variable_value()*

Note: Unlike *get_variable_value()*, this method doesn't look for properties' default values.

set_property_value(*prop*, *value*)
Adds variable with a value for property *prop*.

The property must exist on this model part. This is just a convenience wrapper around *add_variable()* and *get_prop()*.

should_build()
Evaluates the *condition* property on this part and returns True if it should be built or False otherwise. Throws NonConstError if it cannot be determined.

condition
Condition expression (*bkl.expr.Expr*) that describes when should this part of the model be included. If it evaluates to true, the part is build, otherwise it is not. Typical use is enabling some targets or source files only for some toolsets, but it may be more complicated. Depending on the context and the toolset, the expression may even be undeterminable until make-time, if it references some user options (but not all toolsets can handle this). Is `None` if no condition is associated.

module
The *bkl.model.Module* that this part belongs to.

project
The *bkl.model.Project* project this part belongs to.

class bkl.model.**Module**(*parent*, *source_pos*)
Bases: *bkl.model.ModelPart*

Representation of single compilation unit. Corresponds to one Bakefile input file (either specified on command line or imported using *import* command; files included using *include* from other files are *not* represented by Module object) and typically to one generated output file.

targets
Dictionary of all targets defined in this module

source_file
Path to the input `.bkl` source file this module was created from.

srcdir
@srcdir path effective for this module.

imports
Set of filenames of sources imported using the 'import' keyword at this level.

child_parts()
Yields model parts that are (direct) children of this.

enum_props()
Enumerates properties defined on this object.

Like `get_prop()`, this method doesn't work recursively upwards, but lists only properties that are defined for this scope.

See also:

`get_prop()`

is_submodule_of(*module*)
Returns True if this module is (grand-)*child of *module*.

submodules
Submodules of this module.

class bkl.model.**Project**
Bases: *bkl.model.ModelPart*

Abstract model that completely describes state of loaded and processed Bakefile file(s) within the project.

modules
List of all modules included in the project.

configurations
All configurations defined in the project.

settings
List of all settings included in the project.

templates
Dictionary of all templates defined in the project.

add_configuration(*config*)
Adds a new configuration to the project.

add_template(*templ*)
Adds a new template to the project.

all_targets()
Returns iterator over all targets in the project.

child_parts()
Yields model parts that are (direct) children of this.

clone()
Makes an independent copy of the model.

Unlike deepcopy(), this does *not* copy everything, but uses an appropriate mix of deep and shallow copies. For example, expressions, which are read-only, are copied shallowly, but variables or model parts, both of which can be modified in further toolset-specific optimizations, are copied deeply.

enum_props()
Enumerates properties defined on this object.

Like `get_prop()`, this method doesn't work recursively upwards, but lists only properties that are defined for this scope.

See also:

`get_prop()`

get_target(*id*)
Returns Target object identified by its string ID.

has_target(*id*)
Returns true if target with given name exists.

top_module
The toplevel module of this project.

class bkl.model.**ProxyIfResolver**(*config*)
Bases: *bkl.expr.RewritingVisitor*

Replaces references to $(config) with value, allowing the expressions to be evaluated.

if_(*e*)
Called on `IfExpr` expressions.

placeholder(*e*)
Called on `PlaceholderExpr` expressions.

reference(*e*)
Called on `ReferenceExpr` expressions.

class bkl.model.**Setting**(*parent*, *name*, *source_pos*)
Bases: *bkl.model.ModelPart*

User-settable make-time configuration value.

child_parts()
Yields model parts that are (direct) children of this.

enum_props()
Enumerates properties defined on this object.

Like `get_prop()`, this method doesn't work recursively upwards, but lists only properties that are defined for this scope.

See also:

`get_prop()`

class bkl.model.**SourceFile**(*parent*, *filename*, *source_pos*)
Bases: *bkl.model.ModelPart*, *bkl.model.ConfigurationsPropertyMixin*

Source file object.

child_parts()
Yields model parts that are (direct) children of this.

enum_props()
Enumerates properties defined on this object.

Like `get_prop()`, this method doesn't work recursively upwards, but lists only properties that are defined for this scope.

See also:

`get_prop()`

class bkl.model.**Target**(*parent*, *name*, *target_type*, *source_pos*)
Bases: *bkl.model.ModelPart*, *bkl.model.ConfigurationsPropertyMixin*

A Bakefile target.

Variables are typed.

name
Name (ID) of the target. This must be unique in the entire project.

type
Type of the target, as *bkl.api.TargetType* instance.

sources
List of source files, as SourceFile instances.

headers
List of header files, as SourceFile instances. The difference from *sources* is that headers are installable and usable for compilation of other targets, while sources are not.

child_parts()
Yields model parts that are (direct) children of this.

enum_props()
Enumerates properties defined on this object.

Like `get_prop()`, this method doesn't work recursively upwards, but lists only properties that are defined for this scope.

See also:

`get_prop()`

class bkl.model.**Template**(*name*, *bases*, *source_pos=None*)
Bases: `object`

A template.

name
Name of the template.

bases
List of base templates (as *bkl.model.Template* objects).

source_pos
Source code position of template's definition, or `None`.

class bkl.model.**Variable**(*name*, *value*, *type=None*, *readonly=False*, *source_pos=None*)
Bases: `object`

A Bakefile variable.

Variables can be either global or target-specific. Value held by a variable is stored as expression tree, it isn't evaluated into string form until the final stage of processing, when output is generated.

Variables are typed.

name
Name of the variable.

type
Type of the property, as *bkl.vartypes.Type* instance.

value
Value of the variable, as *bkl.expr.Expr* object.

readonly
Indicates if the variable is read-only. Read-only variables can only be assigned once and cannot be modified afterwards.

is_property
Indicates if the variable corresponds to a property.

is_explicitly_set
Indicates if the value was set explicitly by the user. Normally true, only false for properties' default values.

static from_property(*prop*, *value=None*)
Creates a variable from property *prop*. In particular, takes the type and *readonly* attribute from the property. Properties' default value is *not* assigned; *value* is used instead.

set_value(*value*)
Sets new value on the variable. The new value must have same type as current value. Read-only variable cannot be set.

Parameters **value** – New value as *bkl.expr.Expr* object.

bkl.expr – representation of expressions

The *Expr* class that represents a Bakefile expression (be it a condition or variable value) is defined in this module, together with useful functions for evaluating and manipulating expressions.

class bkl.expr.**BoolExpr**(*operator*, *left*, *right=None*, *pos=None*)
Bases: *bkl.expr.Expr*

Boolean expression.

operator
Boolean operator of the node. The value is one of *BoolExpr* constants, e.g. *BoolExpr.AND*.

left
Left (or in the case of NOT operator, only) operand.

right
Right operand. Not set for the NOT operator.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

is_const()

has_bool_operands()
Returns true if the operator is such that it requires boolean operands (i.e. NOT, AND, OR).

AND = '&&'
And operator

EQUAL = '=='
Equality operator

NOT = '!'
Not operator; unlike others, this one is unary and has no right operand.

NOT_EQUAL = '!='
Inequality operator

OR = '||'
Or operator

class bkl.expr.**BoolValueExpr**(*value*, *pos=None*)
Bases: *bkl.expr.Expr*

Constant boolean value, i.e. true or false.

value
Value of the literal, as (Python) boolean.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

is_const()

class bkl.expr.**ConcatExpr**(*items*, *pos=None*)
Bases: *bkl.expr.Expr*

Concatenation of several expression. Typically, used with LiteralExpr and ReferenceExpr to express values such as "$(foo).cpp".

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

is_const()

class bkl.expr.**CondTrackingMixin**
Helper mixin class for tracking currently active condition. Useful for handling e.g. nested if statements.

active_if_cond
Currently active condition, if any.

class bkl.expr.**Expr**(*pos=None*)
Bases: object

Value expression.

Represents a value (typically assigned to a variable, but also expressions used somewhere else, e.g. as conditions) as tree of expression objects. In Bakefile, the expressions are kept in tree representation until the last possible moment, and are manipulated in this form.

Note that expression objects are immutable: if you need to modify an expression, replace it with a new object.

pos
Location of the expression in source tree.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

is_const()

as_symbolic()
Returns the value as a symbolic representation, see SymbolicFormatter.

is_const()
Returns true if the expression is constant, i.e. can be evaluated at this time (bake-time), as opposed to expressions that depend on a setting that can only be determined at make-time.

See also:

as_py()

is_null()
Returns true if the expression evaluates to null, i.e. empty value.

class bkl.expr.**Formatter**(*paths_info*)
Bases: *bkl.expr.Visitor*

Base class for expression formatters. A *formatter* is a class that formats the expression into a string in the way that is needed on the output. For example, it handles things such as path separators on different platforms, variable references, quoting of literals with whitespace in them and so on.

The base class implements commonly used formatting, such as using whitespace for delimiting list items etc.

Use the *format()* method to format an expression.

list_sep
Separator for list items (by default, single space).

paths_info
PathAnchorsInfo information object to use for formatting of paths.

bool(*e*)
Called on *BoolExpr* expressions.

bool_value(*e*)
Called on *BoolValueExpr* expressions.

concat(*e*)
Called on *ConcatExpr* expressions.

format(*e*)
Formats expression *e* into a string.

if_(*e*)
Called on *IfExpr* expressions.

list(*e*)
Called on *ListExpr* expressions.

literal(*e*)
Called on *LiteralExpr* expressions.

null(*e*)
Called on *NullExpr* expressions.

path(*e*)
Called on *PathExpr* expressions.

reference(*e*)
Called on *ReferenceExpr* expressions.

class bkl.expr.**IfExpr**(*cond*, *yes*, *no*, *pos=None*)
Bases: *bkl.expr.Expr*

Conditional expression.

cond
The condition expression.

value_yes
Value of the expression if the condition evaluates to True.

value_no
Value of the expression if the condition evaluates to False.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws

an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

`is_const()`

get_value()
Returns value of the conditional expression, i.e. either *value_yes* or *value_no*, depending on what the condition evaluates to. Throws if the condition cannot be evaluated.

class bkl.expr.**ListExpr**(*items*, *pos=None*)
Bases: *bkl.expr.Expr*

List expression – list of several values of the same type.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

`is_const()`

class bkl.expr.**LiteralExpr**(*value*, *pos=None*)
Bases: *bkl.expr.Expr*

Constant expression – holds a literal.

value
Value of the literal.

pos
Location of the expression in source tree.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

`is_const()`

class bkl.expr.**NullExpr**(*pos=None*)
Bases: *bkl.expr.Expr*

Empty/unset value.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

`is_const()`

class bkl.expr.**PathAnchorsInfo**(*dirsep*, *outfile*, *builddir*, *model*)
Bases: object

Struct with information about real values for symbolic *anchors* of *PathExpr* paths. These are needed in order to format path expression (using *bkl.expr.Formatter* or otherwise).

dirsep
Separator to separate path components ("/" on Unix and "" on Windows).

top_srcdir
Path to the top source directory, as a list of components, relative to the output directory. Will be empty list if the paths are the same.

builddir
Path to the build directory – the directory where object files and other generated files are put – as a list of components, relative to the output directory. Will be empty list if the paths are the same.

top_srcdir_abs
Absolute native path to the top source directory.

outdir_abs
Absolute native path to the output directory – the directory where the project or makefile currently being generated is written to.

builddir_abs
Absolute native path to the build directory.

The constructor creates anchors information from native paths passed to it:

Parameters

- **dirsep** – Path components separator, same as the *dirsep* attribute.
- **outfile** – (Native) path to the output file (project, makefile). Paths in the output will be typically formatted relatively to this path. The *outdir_abs* attribute is computed from this parameter.
- **builddir** – (Native) path to the build directory. May be None if builddir-relative paths make no sense in this context (e.g. for VS2010 solution files).
- **model** – Part of the model (*bkl.model.Module* or *bkl.model.Target*) that *outfile* corresponds to.

class bkl.expr.**PathExpr**(*components*, *anchor='@srcdir'*, *anchor_file=None*, *pos=None*)
Bases: *bkl.expr.Expr*

Expression that holds a file or directory name, or part of it.

components
List of path's components (as expressions). For example, components of path `foo/bar/file.cpp` are `["foo", "bar", "file.cpp"]`.

anchor
The point to which the path is relative to. This can be one of the following:

- `expr.ANCHOR_SRCDIR` – Path is relative to the directory where the input bakefile is (unless overridden in it).
- `expr.ANCHOR_TOP_SRCDIR` – Path is relative to the top source directory (where the toplevel `.bkl` file is, unless overridden in it).
- `expr.ANCHOR_BUILDDIR` – Path is relative to the build directory. Either this anchor or `expr.ANCHOR_TOP_BUILDDIR` should be used for all transient files (object files or other generated files).

- `expr.ANCHOR_TOP_BUILDDIR` – Path is relative to the top build directory. This is currently only used by makefile backends where the build directory for a sub-makefile is different from the top one.

anchor_file
Name of the file the anchor was written in. This is important for @srcdir, which is relative to this place.

pos
Location of the expression in source tree.

as_native_path(*paths_info*)
Returns the path expressed as *absolute* native path. Requires complete *bkl.expr.PathAnchorsInfo* information as its argument.

Throws NonConstError if it cannot be done because of conditional subexpression.

See also:

as_native_path_for_output()

as_native_path_for_output(*model*)
Specialized version of *as_native_path()* that only works with srcdir-based paths. It's useful for code that needs to obtain output file name (which happens *before* PathAnchorsInfo can be constructed).

Parameters **model** – Any part of the model.

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

`is_const()`

change_extension(*newext*)
Changes extension of the filename to *newext* and returns *bkl.expr.PathExpr* with the new path.

get_basename()
Returns basename of the filename path (i.e. the name without directory or extension). Throws if it cannot be determined.

get_directory_path()
Returns PathExpr with the directory component.

get_extension()
Returns extension of the filename path. Throws if it cannot be determined.

is_external_absolute()
Returns true if the path is to be treated as an absolute path provided externally (i.e. via an user setting) when the makefile was invoked.

TODO: add a @absdir anchor instead?

class bkl.expr.**PlaceholderExpr**(*var*, *pos=None*)
Bases: *bkl.expr.Expr*

This is a hack. It is used as placeholder for expressions with not yet known value. In particular, it is used for the "toolset" property before the model is split into toolset-specific copies, to allow partial evaluation common to all of them.

var
Name of referenced setting (e.g. "config" or "toolset").

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

is_const()

class bkl.expr.**ReferenceExpr**(*var*, *context*, *pos=None*)
Bases: *bkl.expr.Expr*

Reference to a variable.

var
Name of referenced variable.

context
Context of the reference, i.e. the scope in which it was used. This is the appropriate *bkl.model.ModelPart* instance (e.g. a target or a module).

as_py()
Returns the expression as Python value (e.g. a list of strings) if it evaluates to a constant literal. Throws an exception if the expression cannot be evaluated at bake-time (such expressions cannot be used in some situations, e.g. to specify output files). Paths are returned as native paths.

Use *bkl.expr.Formatter* if you need to format expressions into strings.

See also:

is_const()

get_value()
Returns value of the referenced variable. Throws an exception if the reference couldn't be resolved.

get_variable()
Return the variable object this reference points to.

Use this method only if necessary; whenever possible, *get_value()* should be used instead.

Note that the returned value may be None, either if the referenced variable doesn't exist or when the reference is to a property that wasn't explicitly set and uses the default value.

class bkl.expr.**RewritingVisitor**
Bases: *bkl.expr.Visitor*

Base class for visitors that perform some changes to the expression.

RewritingVisitor is smart about the rewrites and if nothing changes, returns the same instance, instead of creating an identical copy.

It also implements all Visitor methods to do the right thing by default, so you only need to override the ones that are of interest. Default implementations of the others will do the right thing: for example, if an item in the list is rewritten, the list() method will detect it and return a new list.

bool(*e*)
Called on *BoolExpr* expressions.

concat(*e*)
Called on *ConcatExpr* expressions.

if_(*e*)
Called on *IfExpr* expressions.

list(*e*)
Called on *ListExpr* expressions.

path(*e*)
Called on *PathExpr* expressions.

class bkl.expr.**SymbolicFormatter**
Bases: *bkl.expr.Formatter*

Formats into unambiguous symbolic representation of the expression.

bool(*e*)
Called on *BoolExpr* expressions.

bool_value(*e*)
Called on *BoolValueExpr* expressions.

concat(*e*)
Called on *ConcatExpr* expressions.

if_(*e*)
Called on *IfExpr* expressions.

list(*e*)
Called on *ListExpr* expressions.

literal(*e*)
Called on *LiteralExpr* expressions.

null(*e*)
Called on *NullExpr* expressions.

path(*e*)
Called on *PathExpr* expressions.

placeholder(*e*)
Called on *PlaceholderExpr* expressions.

class bkl.expr.**Visitor**
Bases: object

Implements visitor pattern for *Expr* expressions. This is abstract base class, derived classes must implement all of its methods except *visit()*. The way visitors are used is that the caller calls *visit()* on the expression.

bool(*e*)
Called on *BoolExpr* expressions.

bool_value(*e*)
Called on *BoolValueExpr* expressions.

concat(*e*)
Called on *ConcatExpr* expressions.

if_(*e*)
Called on *IfExpr* expressions.

list(*e*)
Called on *ListExpr* expressions.

literal(*e*)
Called on *LiteralExpr* expressions.

noop(*e*)
Helper to quickly implement handler functions that do nothing.

null(*e*)
Called on *NullExpr* expressions.

path(*e*)
Called on *PathExpr* expressions.

placeholder(*e*)
Called on *PlaceholderExpr* expressions.

reference(*e*)
Called on *ReferenceExpr* expressions.

visit(*e*)
Call this method to visit expression *e*. One of object's "callback" methods will be called, depending on *e*'s type.

Return value is the value returned by the appropriate callback and is typically `None`.

visit_children(*e*)
Helper to implement visitor methods that just need to recursively work on all children. Ignores return value for the children.

bkl.expr.**add_prefix**(*prefix*, *e*)
Adds a *prefix* in front of the expression *e* or, if *e* is a list, in front of each of its elements.

Parameters

- **prefix** – The prefix to add; either an *bkl.expr.Expr* object or a string.
- **e** – The expression to add the prefix in front of.

bkl.expr.**are_equal**(*a*, *b*, *_inside_cond=False*)
Compares two expressions for equality.

Throws the CannotDetermineError exception if it cannot reliably determine equality.

bkl.expr.**concat**(**parts*)
Concatenates all arguments and returns *bkl.expr.ConcatExpr*. The arguments may be expressions or string literals.

bkl.expr.**enum_possible_values**(*e*, *global_cond=None*)
Returns all values that are possible, together with their respective conditions, as an iteratable of (condition, value) tuples. The condition may be `None` if the value is always there, otherwise it is a boolean *bkl.expr.Expr*.

Note that this function returns possible elements for lists. It skips null expressions as well.

Parameters

- **e** – Expression to extract possible values from.
- **global_cond** – Optional condition expression (*bkl.expr.Expr*) to apply to all items. If specified, then every tuple in returned list will have the condition set to either *global_cond* (for unconditional items) or its combination with per-item condition.

bkl.expr.**format_string**(*format*, *values*)
Substitutes occurrences of "%(varname)" in the *format* string with values from the dictionary *values*, similarly to the '%' operator. Values in the dictionary may be either Expr objects or strings.

Note that there's currently no way to escape "%" in such expressions.

bkl.expr.**get_model_name_from_path**(*e*)
Give an expression representing filename, return name suitable for model.SourceFile.name.

bkl.expr.**keep_possible_values_unexpanded**(*e*)
Determines if the given reference value should be expanded or not.

This is a helper of enum_possible_values() and is used to decide whether a value returned by that function is fully expanded or kept in its original form, with variable references in it. The latter is useful when dealing with the source files containing variables as it prevents them from being expanded too early, before the variable value for the current target is really known.

However anything not representing a simple single value should still be expanded and this function task is to check whether the argument is a reference to such simple single value or not.

bkl.expr.**split**(*e*, *sep*)
Splits expression *e* into a list of expressions, using *sep* as the delimiter character. Works with conditional expressions and variable references too.

bkl.expr.**split_into_path**(*e*)
Splits expression *e* into a list of expressions, using '/' as the delimiter character. Returns a PathExpr. Works with conditional expressions and variable references too.

bkl.vartypes – variables types

This module defines types interface as well as basic types. The types – i.e. objects derived from *bkl.vartypes.Type* – are used to verify validity of variable values and other expressions.

class bkl.vartypes.**AnyType**
Bases: *bkl.vartypes.Type*

A fallback type that allows any value at all.

validate(*e*)
Validates if the expression *e* is of this type. If it isn't, throws *bkl.error.TypeError* with description of the error.

Note that this method transparently handles references and conditional expressions.

class bkl.vartypes.**BoolType**
Bases: *bkl.vartypes.Type*

Boolean value type. May be product of a boolean expression or one of the following literals with obvious meanings: "true" or "false".

class bkl.vartypes.**EnumType**(*name*, *allowed_values*)
Bases: *bkl.vartypes.Type*

Enum type. The value must be one of allowed values passed to the constructor.

allowed_values
List of allowed values (strings).

class bkl.vartypes.**IdType**
Bases: *bkl.vartypes.Type*

Type for target IDs.

class bkl.vartypes.**ListType**(*item_type*)
Bases: *bkl.vartypes.Type*

Type for a list of items of homogeneous type.

item_type
Type of items stored in the list (*bkl.vartypes.Type* instance).

class bkl.vartypes.**PathType**
Bases: *bkl.vartypes.Type*

A file or directory name.

class bkl.vartypes.**StringType**
Bases: *bkl.vartypes.Type*

Any string value.

class bkl.vartypes.**Type**
Bases: object

Base class for all Bakefile types.

name
Human-readable name of the type, e.g. "path" or "bool".

normalize(*e*)
Normalizes the expression *e* to be of this type, if it can be done. If it cannot be, does nothing.

Returns *e* if no normalization was done or a new expression with normalized form of *e*.

validate(*e*)
Validates if the expression *e* is of this type. If it isn't, throws *bkl.error.TypeError* with description of the error.

Note that this method transparently handles references and conditional expressions.

bkl.vartypes.**guess_expr_type**(*e*)
Attempts to guess type of the expression if it's possible. Returns AnyType type if unsure.

bkl.vartypes.**normalize_and_validate_bool_subexpressions**(*e*)
Performs type normalization and validation steps for typed subexpressions, namely for IfExpr conditions and boolean expressions.

bkl.vartypes.**TheAnyType = <bkl.vartypes.AnyType object>**
For efficiency, singleton instance of AnyType

bkl.api – public extensions API

class bkl.api.**BuildNode**(*commands*, *inputs=[]*, *outputs=[]*, *name=None*, *source_pos=None*)
Bases: object

BuildNode represents a single node in traditional make-style build graph. Bakefile's model is higher-level than that, its targets may represent entities that will be mapped into several makefile targets. But BuildNode is the simplest element of build process: a list of commands to run together with dependencies that describe when to run it and a list of outputs the commands create.

Node's commands are executed if either a) some of its outputs doesn't exist or b) any of the inputs was modified since the last time the outputs were modified.

See also:

bkl.api.TargetType.get_build_subgraph(), *bkl.api.BuildSubgraph*

name
Name of build node. May be empty. If not empty and the node has no output files (i.e. is *phony*), then this name is used in the generated makefiles. It is ignored in all other cases.

inputs
List of all inputs for this node. Its items are filenames (as *bkl.expr.PathExpr* expressions) or (phony) target names.

outputs
List of all outputs this node generates. Its items are filenames (as *bkl.expr.PathExpr* expressions).

A node with no outputs is called *phony*.

commands:
List of commands to execute when the rebuild condition is met, as *bkl.expr.Expr*.

source_pos
Source code position of whatever code was the cause for the creation of this BuildNode (e.g. associated source file), or `None`.

class bkl.api.**BuildSubgraph**(*main*, *secondary=[]*)
Bases: `object`

BuildSubgraph is a collection of *bkl.api.BuildNode* nodes.

main
Primary build node of a target (e.g. its executable file).

secondary
A list of any secondary nodes needed to build the main nodes (e.g. object files for target's source files). May be empty.

all_nodes()
Yield all nodes included in the subgraph.

class bkl.api.**CustomStep**
Bases: *bkl.api.Extension*

Custom processing step that is applied to the loaded model.

Plugins of this kind can be used to perform any custom operations with the model. For example, they may enforce coding style, they may add or modify some properties or even generate auxiliary output files.

CustomStep class has several hook methods called at different phases of processing. All of them do nothing by default and reimplementing them is optional.

finalize(*model*)
Called after loading the model from file(s) and before doing any optimizations or error checking on it.

Do not create any output files here, leave that to *generate()*.

generate(*model*)
Called right before generating the output. It is called on the common model, before optimizing it for individual toolsets.

It is permitted to create output files in this method.

class bkl.api.**Extension**
Bases: `object`

Base class for all Bakefile extensions.

Extensions are singletons, there's always only one instance of given extension at runtime. Use the get() method called on appropriate extension type to obtain it. For example:

program = TargetType.get("program") # . . . do something with it. . .

name
Use-visible name of the extension. For example, the name for targets extensions is what is used in target declarations; likewise for property names.

classmethod all()
Returns iterator over instances of all implementations of this extension type.

classmethod all_names()
Returns names of all implementations of this extension type.

classmethod all_properties(*kind='properties'*)
For derived extension types that have properties e.g. *TargetType*), returns iterator over all properties.

The class must have *class* member variable var:*properties* with a list of *bkl.api.Property* instances. Base class' properties are automagically scanned too.

Parameters **kind** – Kind (i.e. attribute name) of the properties to list. By default, "properties", but may be more specific, e.g. "properties_module" or "properties_vs2010".

See also:

bkl.api.Property

classmethod all_properties_kinds()
Returns a set of names of all properties attributes in this class. These are attributes named "properties" or "properties_<something>", e.g. "properties_module" for properties with module scope.

classmethod get(*name=None*)
This class method is used to get an instance of an extension. In can be used in one of two ways:

1. When called on an extension type class with *name* argument, it returns instance of extension with given name and of the extension type on which this classmethod was called:

```
>>> bkl.api.Toolset.get("gnu")
    <bkl.plugins.gnu.GnuToolset object at 0x2232950>
```

2. When called without the *name* argument, it must be called on particular extension class and returns its (singleton) instance:

```
>>> GnuToolset.get()
    <bkl.plugins.gnu.GnuToolset object at 0x2232950>
```

Parameters **name** – Name of the extension to read; this corresponds to class' "name" attribute. If not specified, then get() must be called on a extension, not extension base class.

class bkl.api.**FileCompiler**
Bases: *bkl.api.Extension*

In Bakefile API, FileCompiler is used to define all compilation steps.

Traditionally, the term *compiler* is used for a tool that compiles source code into object files. In Bakefile, a *file compiler* is generalization of this term: it's a tool that compiles file or files of one object type into one or more files of another type. In this meaning, a C/C++ compiler is a *file compiler*, but so is a linker (it "compiles" object files into executables) or e.g. Lex/Yacc compiler or Qt's MOC preprocessor.

commands(*toolset*, *target*, *input*, *output*)
Returns list of commands (as *bkl.expr.Expr*) to invoke the compiler.

Parameters

- **toolset** – Toolset used.
- **target** – The target object for which the invocation is done.
- **input** – Input file (*bkl.expr.PathExpr*) or files (*bkl.expr.ListExpr*), depending on cardinality.
- **output** – *bkl.expr.Expr* expression with the name of output file.

is_supported(*toolset*)
Returns whether given toolset is supported by this compiler.

Default implementation returns True for all toolsets.

cardinality = '1'
Cardinality of the compiler. That is, whether it compiles one file into one file (`FileCompiler.ONE_TO_ONE`, e.g. C compilers) or whether it compiles many files of the same type into one output file (`FileCompiler.MANY_TO_ONE`, e.g. the linker or Java compiler).

in_type = None
bkl.api.FileType for compiler's input file.

out_type = None
bkl.api.FileType for compiler's output file.

class bkl.api.**FileRecognizer**
Bases: `object`

Mixin base class for extensions that handle certain file types and need to be associated with a file automatically. The class provides easy to use get_for_file() mechanism.

To use this class, derive from both *bkl.api.Extension* and this one.

detect(*filename*)
Returns True if the file *filename* is supported by the class. Note that it is only called if the file has one of the extensions listed in *extensions*. By default, returns `True`.

Parameters **filename** – Name of the file to check. Note that this is native filename (as a string) and points to existing file.

classmethod get_for_file(*filename*)
Returns appropriate implementation of the class for given file.

Throws *bkl.error.UnsupportedError* if no implementation could be found.

Parameters **filename** – Name of the file, as a native path.

extensions = []
List of file extensions recognized by this extension, without dots (e.g. `["vcproj", "vcxproj"]`).

class bkl.api.**FileType**(*extensions=[]*)
Bases: *bkl.api.Extension*, *bkl.api.FileRecognizer*

Description of a file type. File types are used by *bkl.api.FileCompiler* to define both input and output files.

extensions
List of extensions for this file type, e.g. `["cpp", "cxx", "cc"]`.

class bkl.api.**Property**(*name*, *type*, *default=None*, *readonly=False*, *inheritable=False*, *doc=None*)
Bases: `object`

Properties describe variables on targets etc. that are part of the API – that is, they have special meaning for the toolset and. Unlike free-form variables, properties have a type associated with them and any values assigned to them are checked for type correctness. They can optionally have a default value, too.

name
Name of the property/variable.

type
Type of the property, as *bkl.vartypes.Type* instance.

default
Default value of the property (as *bkl.expr.Expr* or a function that returns an expression) or `None`. If not specified (i.e. `None`), then this property is required and must always be set to a value in the bakefile.

readonly
Indicates if the property is read-only. Read-only properties can only have the default value and cannot be modified. They are typically derived from some other value and exist as a convenience. An example of read-only property is the `id` property on targets.

scopes
Optional scope of the property, as list of strings. Each item may be one of `Property.SCOPE_PROJECT`, `Property.SCOPE_MODULE`, `Property.SCOPE_TARGET` for any target or target type name (e.g. `program`) for scoping on specific target name.

Finally, may be `None` for default (depending from where the property was obtained from).

inheritable
A property is *inheritable* if its value can be specified in the (grand-)parent scope. For example, an inheritable property on a target may be specified at the module level or even in the parent module; an inheritable property on a source file (e.g. "defines") may be specified on the target, the module and so on.

toolsets
List of toolset names for toolsets this property applies to. This is mostly for documentation purposes, it doesn't affect their processing. Is `None` for toolset-agnostic properties.

doc
Optional documentation for the property.

Example usage:

```
class FooTarget(bkl.api.TargetType):
    name = "foo"
    properties = [
        Property("deps",
                 type=ListType(IdType()),
                 default=[],
                 doc="Target's dependencies (list of IDs).")
    ]
    ...
```

default_expr (*for_obj*, *throw_if_required*)
Returns the value of *default* expression. Always returns an *bkl.expr.Expr* instance, even if the default is of a different type.

Parameters

- **for_obj** – The class:*bkl.model.ModelPart* object to return the default for. If the default value is defined, its expression is evaluated in the context of *for_obj*.
- **throw_if_required** – If False, returns NullExpr if the property is a required one (doesn't have a default value). If True, throws in that case.

internal
True if the property is for internal purposes and shouldn't be used by users, False otherwise.

class bkl.api.**TargetType**
Bases: *bkl.api.Extension*

Base class for implementation of a new target type.

get_build_subgraph (*toolset*, *target*)
Returns a *bkl.api.BuildSubgraph* object with description of this target's local part of build graph – that is, its part needed to produce output files associated with this target.

Usually, a BuildSubgraph with just one main BuildNode will be returned, but it's possible to have TargetTypes that correspond to more than one makefile target (e.g. libtool-style libraries or gettext catalogs).

Parameters

- **toolset** – The toolset used (*bkl.api.Toolset*).
- **target** – Target instance (*bkl.model.Target*).

vs_project (*toolset*, *target*)
Returns Visual Studio project file object (derived from `bkl.plugins.vsbase.VSProjectBase`) if the target type can be implemented as a Visual Studio project.

Implementing this method is strictly optional.

properties = []
List of all properties supported on this target type, as *Property* instances. Note that properties list is automagically inherited from base classes, if any.

class bkl.api.**Toolset**
Bases: *bkl.api.Extension*

This class encapsulates generating of the project files or makefiles.

The term "toolset" refers to collection of tools (compiler, linker, make, IDE, ...) used to compile programs. For example, "Visual C++ 2008", "Visual C++ 2005", "Xcode" or "Borland C++" are toolsets.

In Bakefile API, this class is responsible for creation of the output. It puts all the components (platform-specific commands, make syntax, compiler invocation, ...) together and writes out the makefiles or projects.

generate (*project*)
Generates all output files for this toolset.

Parameters **project** – model.Project instance with complete description of the output. It was already preprocessed to remove content not relevant for this toolset (e.g. targets or submodules built conditionally only for other toolsets, conditionals that are always true or false within the toolset and so on).

get_builddir_for (*target*)
Returns build directory used for *target*.

Returned value must be a *bkl.expr.PathExpr* expression object, but it doesn't have to be a constant (for example, it may reference configuration name, as e.g. Visual Studio does).

The function is called after the model is fully loaded and partially simplified, on a model already specialized for this toolset only. It is used to replace path expression with the relative *@builddir* anchor with absolute paths.

properties = []
List of all properties supported on this target type, as *Property* instances. Note that properties list is automagically inherited from base classes, if any.

bkl.error – exceptions and errors handling

This module contains helper classes for simple handling of errors. In particular, the *Error* class keeps track of the position in source code where the error occurred or to which it relates to.

exception bkl.error.**CannotDetermineError**(*msg=None*, *pos=None*)
Bases: *bkl.error.NonConstError*

Exception thrown when something (e.g. equality) cannot be determined. This usually signifies a weakness in Bakefile implementation that should be improved.

exception bkl.error.**Error**(*msg*, *pos=None*)
Bases: exceptions.Exception

Base class for all Bakefile errors.

When converted to string, the message is formatted in the usual way of compilers, as `file:line: error`.

msg
Error message to show to the user.

pos
bkl.parser.ast.Position object with location of the error. May be `None`.

exception bkl.error.**NonConstError**(*msg*, *pos=None*)
Bases: *bkl.error.Error*

Exception thrown when attempting to convert an expression into bake-time constant.

exception bkl.error.**NotFoundError**(*msg*, *pos=None*)
Bases: *bkl.error.Error*

Exception thrown when a property or variable wasn't found at all.

exception bkl.error.**ParserError**(*msg*, *pos=None*)
Bases: *bkl.error.Error*

Exception class for errors encountered by the Bakefile parser.

exception bkl.error.**TypeError**(*type*, *expr*, *msg=None*, *pos=None*)
Bases: *bkl.error.Error*

Exception class for variable type errors.

detail
Any extra details about the error or (usually) `None`.

See also:

bkl.vartypes.Type

Convenience constructor creates error message appropriate for the type and expression test, in the form of `expression expr is not type` or `expression expr is not type: msg` if additional message is supplied.

Parameters

- **type** – *bkl.vartypes.Type* instance the error is related to.
- **expr** – *bkl.expr.Expr* expression that caused the error.
- **msg** – Optional error message detailing reasons for the error. This will be stored as *detail* if provided.

exception bkl.error.**UndefinedError**(*msg*, *pos=None*)
Bases: *bkl.error.Error*

Exception thrown when a property or variable is undefined, i.e. doesn't have a value.

exception bkl.error.**UnsupportedError**(*msg*, *pos=None*)
Bases: *bkl.error.Error*

Exception class for errors when something is unsupported, e.g. unrecognized file extension.

exception bkl.error.**VersionError**(*msg*, *pos=None*)
Bases: *bkl.error.Error*

Exception raised when Bakefile version is too old for the input.

class bkl.error.**error_context**(*context*)
Error context for adding positional information to exceptions thrown without one. This can happen in some situations when the particular expression causing the error isn't available. In such situations, it's much better to provide coarse position information (e.g. a target) instead of not providing any at all.

Usage:

```
with error_context(target):
    ...do something that may throw...
```

pos
bkl.parser.ast.Position object with location of the error. May be None.

bkl.error.**warning**(*msg*, **args*, ***kwargs*)
Logs a warning.

The function takes position arguments similarly to logging module's functions. It also accepts options *pos* argument with position information as *bkl.parser.ast.Position*.

Uses active *error_context* instances to decorate the warning with position information if not provided.

Usage:

```
bkl.error.warning("target %s not supported", t.name, pos=t.source_pos)
```

bkl.io – I/O helpers

Helper classes for Bakefile I/O. Manages atomic writing of output, detecting changes, line endings conversions etc.

class bkl.io.**OutputFile**(*filename*, *eol*, *charset='utf-8'*, *creator=None*, *create_for=None*)
Bases: object

File to be written by Bakefile.

Example usage:

```
f = io.OutputFile("Makefile")
f.write(body)
f.commit()
```

Notice the need to explicitly call commit().

Creates output file.

Parameters

- **filename** – Name of the output file. Should be either relative to CWD or absolute; the latter is recommended.
- **eol** – Line endings to use. One of EOL_WINDOWS and EOL_UNIX.
- **charset** – Charset to use if Unicode string is passed to write().

- **creator** – Who is creating the file; typically toolset object.
- **create_for** – Object the file is created for, e.g. a module or a target.

replace(*placeholder*, *value*)
Replaces the value of the given placeholder with its real value. This is useful for parts of the output which are not known at the time they are written because they depend on other parts coming after them.

Notice that only the first occurrency of the placeholder is replaced.

write(*text*)
Writes text to the output, performing line endings conversion as needed. Note that the changes don't take effect until you call commit().

bkl.makefile – support for implementing makefiles toolsets

Foundation code for makefile-based toolsets.

All makefile-based toolsets should derive from MakefileToolset defined in this module.

class bkl.makefile.**MakefileExprFormatter**(*toolset*, *paths_info*)
Bases: *bkl.expr.Formatter*

literal(*e*)
Called on `LiteralExpr` expressions.

path(*e*)
Called on `PathExpr` expressions.

placeholder(*e*)
Called on `PlaceholderExpr` expressions.

class bkl.makefile.**MakefileFormatter**
Bases: *bkl.api.Extension*

MakefileFormatter extensions are used to format makefiles content (i.e. targets and their commands) in the particular makefiles format.

This includes things such as expressing conditional content, referencing variables and so on.

Note that formatters do *not* handle platform- or compiler-specific things, e.g. path separators or compiler invocation. There are done by *bkl.expr.Formatter* and *bkl.api.FileCompiler* classes.

This base class implements methods that are common for most make variants; derived classes can override them and they must implement the rest.

comment(*text*)
Returns given (possibly multi-line) string formatted as a comment.

Parameters **text** – text of the comment

multifile_target(*outputs*, *outfiles*, *deps*, *commands*)
Returns string with target definition for targets that produce multiple files. A typical example is Bison parser generator, which produces both .c and .h files.

Parameters

- **outputs** – List of output files of the rule, as objects.
- **outfiles** – List of output files of the rule, as strings.
- **deps** – See target()
- **commands** – See target()

submake_command(*directory*, *filename*, *target*)
Returns string with command to invoke `make` in subdirectory *directory* on makefile *filename*, running *target*.

target(*name*, *deps*, *commands*)
Returns string with target definition.

Parameters

- **name** – Name of the target.
- **deps** – List of its dependencies. Items are strings corresponding to some target's name (may be expressions that reference a variable, in that case the string must already be formatted with appropriate *bkl.expr.Formatter*). May be empty.
- **commands** – List of commands to execute to build the target; they are already formatted to be in make's syntax and each command in the list is single-line shell command. May be `None`.

var_definition(*var*, *value*)
Returns string with definition of a variable value, typically *var* = *value*.

Parameters

- **var** – variable being defined
- **value** – value of the variable; this string is already formatted to be in make's syntax and may be multi-line

class bkl.makefile.**MakefileToolset**
Bases: *bkl.api.Toolset*

Base class for makefile-based toolsets.

ExprFormatter
`expr.Formatter`-derived class for this toolset.

alias of *MakefileExprFormatter*

generate(*project*)
Generates all output files for this toolset.

Parameters **project** – model.Project instance with complete description of the output. It was already preprocessed to remove content not relevant for this toolset (e.g. targets or submodules built conditionally only for other toolsets, conditionals that are always true or false within the toolset and so on).

get_builddir_for(*target*)
Returns build directory used for *target*.

Returned value must be a *bkl.expr.PathExpr* expression object, but it doesn't have to be a constant (for example, it may reference configuration name, as e.g. Visual Studio does).

The function is called after the model is fully loaded and partially simplified, on a model already specialized for this toolset only. It is used to replace path expression with the relative @*builddir* anchor with absolute paths.

on_footer(*file*, *module*)
Called at the end of generating the output to add any ending text, for example unconditional inclusion of dependencies tracking code.

on_header(*file*, *module*)
Called before starting generating the output to add any header text, typically used to pre-define any make variables.

Call the base class version first to insert a warning about the file being auto-generated.

on_phony_targets (*file*, *targets*)
Called with a list of all phony (i.e. not producing actual files) targets (as their names as strings) when generating given file.

Formatter = None
MakefileFormatter-derived class for this toolset.

autoclean_extensions = []
Files with extensions from this list will be automatically deleted by "make clean".

default_makefile = None
Default filename from output makefile.

del_command = None
Command used to delete files

bkl.compilers – FileCompiler helpers

Helpers for working with *bkl.api.FileType* and *bkl.api.FileCompiler* extensions.

class bkl.compilers.**CFileType**
Bases: *bkl.api.FileType*

class bkl.compilers.**CxxFileType**
Bases: *bkl.api.FileType*

class bkl.compilers.**NativeLibFileType**(*extensions=[]*)
Bases: *bkl.api.FileType*

class bkl.compilers.**NativeLoadableModuleFileType**(*extensions=[]*)
Bases: *bkl.api.FileType*

class bkl.compilers.**NativeProgramFileType**(*extensions=[]*)
Bases: *bkl.api.FileType*

class bkl.compilers.**NativeSharedLibraryFileType**(*extensions=[]*)
Bases: *bkl.api.FileType*

bkl.compilers.**disambiguate_intermediate_file_names**(*files*)
Given a list of SourceFile objects, finds files that would have conflicting object file names (e.g. foo/x.cpp and bar/x.cpp would use the same x.obj filename).

Returns dictionary with SourceFile objects as keys and unambiguous basenames (e.g. 'x_foo' and 'x_bar' for the above example). Only files with conflicts are included in the dictionary (consequently, it will be empty or near-empty most of the time).

bkl.compilers.**get_compilation_subgraph**(*toolset*, *target*, *ft_to*, *outfile*)
Given list of source files (as *bkl.expr.ListExpr*), produces build subgraph (*bkl.api.BuildSubgraph*) with appropriate *bkl.api.BuildNode* nodes.

Parameters

- **toolset** – The toolset used (as *bkl.api.Toolset*).
- **target** – The target object for which the invocation is done.
- **ft_to** – Type of the output file to compile to.
- **outfile** – Name of the output file (as *bkl.expr.PathExpr*).

bkl.compilers.**get_compiler**(*toolset*, *ft_from*, *ft_to*)
Finds the compiler that compiles files of type *ft_from* into *ft_to*. Both arguments are *bkl.api.FileType* instances.

The returned object is a singleton. If such compiler cannot be found, returns None.

bkl.compilers.**get_file_type**(*extension*)
Returns file type instance based on extension.

The returned object is a singleton.

```
>>> a = get_file_type("cpp")
>>> b = get_file_type("cpp")
>>> assert a is b
```

bkl.compilers.**get_file_types_compilable_into**(*toolset*, *ft*)
Returns file types that can be compiled into *ft*.

5.2.2 Internals reference documentation

bkl – bootstrapping Bakefile

bkl.parser – language parser

bkl.parser.**get_parser**(*code*, *filename=None*)
Prepares Bakefile parser for parsing given Bakefile code from string argument passed in. The optional filename argument allows specifying input file name for the purpose of errors reporting.

bkl.parser.**parse**(*code*, *filename=None*, *detect_compatibility_errors=True*)
Reads Bakefile code from string argument passed in and returns parsed AST. The optional filename argument allows specifying input file name for the purpose of errors reporting.

bkl.parser.ast – AST representation

class bkl.parser.ast.**AndNode**(*payload*)
Bases: *bkl.parser.ast.BoolNode*

class bkl.parser.ast.**AppendNode**(*payload*)
Bases: *bkl.parser.ast.AssignmentNode*

Assignment of value to a variable by appending (operator +=).

class bkl.parser.ast.**AssignmentNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Assignment of value to a variable.

lvalue
Variable assigning to, LvalueNode

value
Value being assigned.

class bkl.parser.ast.**BaseListNode**(*payload*)
Bases: *bkl.parser.ast.Node*

List of base templates.

names
List of strings with base names

class bkl.parser.ast.**BoolNode**(*payload*)
Bases: *bkl.parser.ast.Node*

left
Left operand

operator
Boolean operator (token type, e.g. AND)

right
Right operand

class bkl.parser.ast.**BoolvalNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Boolean constant (true/false).

value
Value of the node, as boolean

class bkl.parser.ast.**ConcatNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Concatenation of several parts, to form single string.

values
List of fragments.

class bkl.parser.ast.**ConfigurationNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Definition of a configuration.

base
Name of the base configuration or None

content
Other content: variables assignments and such

name
Name of the configuration

class bkl.parser.ast.**EqualNode**(*payload*)
Bases: *bkl.parser.ast.BoolNode*

class bkl.parser.ast.**FilesListNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Setting of sources/headers.

files
List of files.

kind
Sources/headers

class bkl.parser.ast.**IdNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Identifier (variable, target, template, . . .).

class bkl.parser.ast.**IfNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Conditional content node – "if" statement.

cond
Condition expression

content
Conditional statements

class bkl.parser.ast.**ImportNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Textual inclusion of a file.

file
File to include

class bkl.parser.ast.**ListNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Right side of variable assignment, contains list of values (LiteralNode, VarReferenceNode etc.).

values
List of values in the assignment. May be single value, maybe be multiple values, code using this must correctly interpret it and check values' types.

class bkl.parser.ast.**LiteralNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Single value, i.e. literal.

text
Text of the value, as string.

class bkl.parser.ast.**LvalueNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Left side of assignment.

scope
List of scope identifiers; first one may be None for global.

var
Variable assigning to

class bkl.parser.ast.**NilNode**(*payload=None*)
Bases: *bkl.parser.ast.Node*

Empty node.

class bkl.parser.ast.**Node**(*payload*)
Bases: antlr3.tree.CommonTree

Base class for Bakefile AST tree node.

toString()
Override to say how a node (not a tree) should look as text

toStringTree(*indent=''*)
Print out a whole tree not just a node

class bkl.parser.ast.**NotEqualNode**(*payload*)
Bases: *bkl.parser.ast.BoolNode*

class bkl.parser.ast.**NotNode**(*payload*)
Bases: *bkl.parser.ast.BoolNode*

class bkl.parser.ast.**OrNode**(*payload*)
Bases: *bkl.parser.ast.BoolNode*

class bkl.parser.ast.**PathAnchorNode**(*payload*)
Bases: *bkl.parser.ast.LiteralNode*

A literal with path anchor (@srcdir etc.).

class bkl.parser.ast.**PluginNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Inclusion of a plugin.

file
File with plugin code

class bkl.parser.ast.**Position**(*filename=None*, *line=None*, *column=None*)
Bases: object

Location of an error in input file.

All of its attributes are optional and may be None. Convert the object to string to get human-readable output.

filename
Name of the source file.

line
Line number.

column
Column on the line.

class bkl.parser.ast.**RootNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Root node of loaded .bkl file.

class bkl.parser.ast.**SettingNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Definition of a user setting.

content
Properties assignments and such

name
Name of the setting

class bkl.parser.ast.**SrcdirNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Overriding of the @srcdir value.

srcdir
The new srcdir directory

class bkl.parser.ast.**SubmoduleNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Inclusion of a submodule.

file
File with submodule definition

class bkl.parser.ast.**TargetNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Creation of a makefile target.

base_templates
List of names of base templates

content
Other content: variables assignments and such

name
Name of the target

type
Type of the target

class bkl.parser.ast.**TemplateNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Template definition node.

base_templates
List of names of base templates

content
Other content: variables assignments and such

name
Name of the target

class bkl.parser.ast.**VarReferenceNode**(*payload*)
Bases: *bkl.parser.ast.Node*

Reference to a variable.

var
Referenced variable

bkl.interpreter – language interpreter

This module contains the very core of Bakefile – the interpreter, *bkl.interpreter.Interpreter*, and its supporting classes.

class bkl.interpreter.**Interpreter**
Bases: object

The interpreter is responsible for doing everything necessary to "translate" input `.bkl` files into generated native makefiles. This includes building a project model from the input, checking it for correctness, optimizing it and creating outputs for all enabled toolsets.

Interpreter provides both high-level interface for single-call usage (see *process()*) and other methods with finer granularity that allows you to inspect individual steps (most useful for the test suite).

model
Model of the project, as *bkl.model.Project*. It's state always reflects current state of processing.

toolsets_to_use
Set of toolsets to generate for. This list may contain only a subset of toolsets the bakefile is written for and may even contain toolsets not specified in the bakefile.

If `None` (the default), then the toolsets listed in the bakefile are used.

add_module(*ast*, *parent*)
Adds parsed AST to the model, without doing any optimizations. May be called more than once, with different parsed files.

Parameters **ast** – AST of the input file, as returned by `bkl.parser.parse_file()`.

finalize()
Finalizes the model, i.e. checks it for validity, optimizes, creates per-toolset models etc.

finalize_for_toolset(*toolset_model*, *toolset*)
Finalizes after "toolset" variable was set.

generate()
Generates output files.

generate_for_toolset(*toolset*, *skip_making_copy=False*)
Generates output for given *toolset*.

limit_toolsets(*toolsets*)
Sets *toolsets_to_use*.

make_toolset_specific_model(*toolset*, *skip_making_copy=False*)
Returns toolset-specific model, i.e. one that works only with *toolset*, has the `toolset` property set to it. The caller still needs to call finalize_for_toolset() on it.

process(*ast*)
Interprets input file and generates the outputs.

Parameters **ast** – AST of the input file, as returned by `bkl.parser.parse_file()`.

Processing is done in several phases:

1. Basic model is built (see *bkl.interpreter.builder.Builder*). No optimizations or checks are performed at this point.
2. Several generic optimization and checking passes are run on the model. Among other things, types correctness and other constraints are checked, variables are substituted and evaluated.
3. The model is split into several copies, one per output toolset.
4. Further optimization passes are done.
5. Output files are generated.

Step 1 is done by *add_module()*. Steps 2-4 are done by *finalize()* and step 5 is implemented in *generate()*.

process_file(*filename*)
Like *process()*, but takes filename as its argument.

class bkl.interpreter.builder.**Builder**(*on_submodule=None*)
Bases: `object`, *bkl.expr.CondTrackingMixin*

interpreter.Builder processes parsed AST and builds a project model from it.

It doesn't do anything smart like optimizing things, it does only the minimal processing needed to produce a valid, albeit suboptimal, model.

This includes checking variables scopes etc., but does *not* involve checks for type correctness. Passes further in the *bkl.interpreter.Interpreter* pipeline handle that.

context
Current context. This is the inner-most *bkl.model.ModelPart* at the time of parsing. Initially, it is set to a new *bkl.model.Module* instance by *create_model()*. When descending into a target, it is temporarily set to said target and then restored and so on.

Constructor.

Parameters **on_module** – Callback to call (with filename as argument) on `submodule` statement.

create_expression(*ast*, *parent*)
Creates `bkl.epxr.Expr` expression in given parent's context.

create_model(*ast*, *parent*)
Returns constructed model, as *bkl.model.Module* instance.

handle_children(*children*, *context*)
Runs model creation of all children nodes.

Parameters

- **children** – List of AST nodes to treat as children.
- **context** – Context (aka "local scope"). Interpreter's *context* is set to it for the duration of the call.

CHAPTER 6

Glossary

bake-time The moment when the Bakefile tool `bkl` is run to *create* native makefiles and projects. Nothing is compiled at this time, only the makefiles are created. Compare with *make-time*.

make-time The moment when a generated makefile, project or solution is *used* by the user to build their project. Compare with *bake-time*.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

C

D

E

F

N

O

P

R

S

T

U

V

W