
Boost.Jam : 3.1.18

Rene Rivera
David Abrahams
Vladimir Prus

Copyright © 2003-2007 Rene Rivera, David Abrahams, Vladimir Prus

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Introduction	1
Building BJam	2
Using BJam	6
Options	6
Operation	8
Language	9
Lexical Features	9
Targets	9
Rules	10
Flow-of-Control	17
Variables	19
Modules	24
Miscellaneous	27
Diagnostics	27
Bugs, Limitations	28
Fundamentals	28
History	29

Introduction



Warning

Most probably, you are looking for [Boost.Build manual](#). This document is not meant to be read standalone and will only confuse you. Boost.Build manual refers to specific sections when necessary.

Boost.Jam (BJam) is the low-level build engine tool for [Boost.Build](#). Historically, Boost.Jam is based on on FTJam and on [Perforce Jam](#) but has grown a number of significant features and is now developed independently, with no merge back expected to happen, and little use outside Boost.Build.

This is version 3.1.18 of BJam and is based on version 2.4 of Jam/MR:

```
+\  
+\  
  Copyright 1993-2002 Christopher Seiwald and Perforce Software, Inc.  
+\  
+\  
This is Release 2.4 of Jam/MR, a make-like program.  
License is hereby granted to use this software and distribute it  
freely, as long as this copyright notice is retained and modifications  
are clearly marked.  
ALL WARRANTIES ARE HEREBY DISCLAIMED.
```

Building BJam

Installing BJam after building it is simply a matter of copying the generated executables someplace in your `PATH`. For building the executables there are a set of `build` bootstrap scripts to accomodate particular environments. The scripts take one optional argument, the name of the toolset to build with. When the toolset is not given an attempt is made to detect an available toolset and use that. The build scripts accept these arguments:

```
build [toolset]
```

Running the scripts without arguments will give you the best chance of success. On Windows platforms from a command console do:

```
cd jam source location
.\build.bat
```

On Unix type platforms do:

```
cd jam source location
sh ./build.sh
```

For the Boost.Jam source included with the Boost distribution the *jam source location* is BOOST_ROOT/tools/jam/src.

If the scripts fail to detect an appropriate toolset to build with your particular toolset may not be auto-detectable. In that case, you can specify the toolset as the first argument, this assumes that the toolset is readily available in the `PATH`.



Note

The toolset used to build Boost.Jam is independent of the toolsets used for Boost.Build. Only one version of Boost.Jam is needed to use Boost.Build.

The supported toolsets, and whether they are auto-detected, are:

Table 1. Supported Toolsets

Script	Platform	Toolset	Detection and Notes
build.bat	Windows NT, 2000, and XP	borland Borland C++Builder (BCC 5.5)	<ul style="list-style-type: none"> • Common install location: "C:\Borland\BCC55" • BCC32.EXE in PATH
		como Comeau Computing C/C++	
		gcc GNU GCC	
		gcc-nocygwin GNU GCC	
		intel-win32 Intel C++ Compiler for Windows	<ul style="list-style-type: none"> • ICL.EXE in PATH
		metrowerks MetroWerks CodeWarrior C/C++ 7.x, 8.x, 9.x	<ul style="list-style-type: none"> • CWFoldervariable configured • MWCC.EXE in PATH
		mingw GNU GCC as the MinGW configuration	<ul style="list-style-type: none"> • Common install location: "C:\MinGW"
		msvc Microsoft Visual C++ 6.x	<ul style="list-style-type: none"> • VCVARS32.BAT already configured • %MSVCDIR% is present in environment • Common install locations: "%ProgramFiles%\Microsoft Visual Studio", "%ProgramFiles%\Microsoft Visual C++" • CL.EXE in PATH
		vc7 Microsoft Visual C++ 7.x	<ul style="list-style-type: none"> • VCVARS32.BAT or VSvars32.BAT already configured • %VS71COMNTOOLS% is present in environment • %VCINSTALLDIR% is present in environment • Common install locations: "%ProgramFiles%\Microsoft Visual Studio .NET", "%ProgramFiles%\Microsoft Visual Studio .NET 2003" • CL.EXE in PATH

Script	Platform	Toolset	Detection and Notes
		vc8 and vc9 Microsoft Visual C++ 8.x and 9.x	Detection: <ul style="list-style-type: none"> VCVARSALL.BAT already configured %VS90COMNTOOLS% is present in environment Common install location: "%ProgramFiles%\Microsoft Visual Studio 9" %VS80COMNTOOLS% is present in environment Common install location: "%ProgramFiles%\Microsoft Visual Studio 8" CL.EXE in PATH Notes: <ul style="list-style-type: none"> If VCVARSALL.BAT is called to set up the toolset, it is passed all the extra arguments, see below for what those arguments are. This can be used to build, for example, a Win64 specific version of bjam. Consult the VisualStudio documentation for what the possible argument values to the VCVARSALL.BAT are.
build.sh	Unix, Linux, Cygwin, etc.	acc HP-UX aCC	<ul style="list-style-type: none"> aCC in PATH uname is "HP-UX"
		como Comeau Computing C/C++	<ul style="list-style-type: none"> como in PATH
		gcc GNU GCC	<ul style="list-style-type: none"> gcc in PATH
		intel-linux Intel C++ for Linux	<ul style="list-style-type: none"> icc in PATH Common install locations: "/opt/intel/cc/9.0", "/opt/intel_cc_80", "/opt/intel/compiler70", "/opt/intel/compiler60", "/opt/intel/compiler50"
		kcc Intel KAI C++	<ul style="list-style-type: none"> KCC in PATH
		kylix Borland C++Builder	<ul style="list-style-type: none"> bc++ in PATH
		mipspro SGI MIPSpro C	<ul style="list-style-type: none"> uname is "IRIX" or "IRIX64"
		sunpro Sun Workshop 6 C++	<ul style="list-style-type: none"> Standard install location: "/opt/SUNWspro"

Script	Platform	Toolset	Detection and Notes
		qcc QNX Neutrino	<ul style="list-style-type: none"> • uname is "QNX" and qcc in PATH
		true64cxx Compaq C++ Compiler for True64 UNIX	<ul style="list-style-type: none"> • uname is "OSF1"
		vacpp IBM VisualAge C++	<ul style="list-style-type: none"> • xlc in PATH
	MacOS X	darwin Apple MacOS X GCC	<ul style="list-style-type: none"> • uname is "Darwin"
	Windows NT, 2000, and XP	mingw GNU GCC as the MinGW configuration with the MSYS shell	<ul style="list-style-type: none"> • Common install location: "/mingw"

The built executables are placed in a subdirectory specific to your platform. For example, in Linux running on an Intel x86 compatible chip, the executables are placed in: "bin.linuxx86". The `=bjam[.exe]` executable can be used to invoke Boost.Build.

The build scripts support additional invocation arguments for use by developers of Boost.Jam and for additional setup of the toolset. The extra arguments come after the toolset:

- Arguments not in the form of an option, before option arguments, are used for extra setup to toolset configuration scripts.
- Arguments of the form "`--option`", which are passed to the `build.jam` build script.
- Arguments not in the form of an option, after the options, which are targets for the `build.jam` script.

```
build [toolset] [setup*] [--option+ target*]
```

The arguments immediately after the toolset are passed directly to the setup script of the toolset, if available and if it needs to be invoked. This allows one to configure the toolset as needed to do non-default builds of `bjam`. For example to build a Win64 version with `vc8`. See the toolset descriptions above for when particular toolsets support this.

The arguments starting with the "`--option`" forms are passed to the `build.jam` script and are used to further customize what gets built. Options and targets supported by the `build.jam` script:

<code>---</code>	Empty option when one wants to only specify a target.
<code>--release</code>	The default, builds the optimized executable.
<code>--debug</code>	Builds debugging versions of the executable. When built they are placed in their own directory "bin./platform/.debug".
<code>--grammar</code>	Normally the Jam language grammar parsing files are not regenerated. This forces building of the grammar, although it may not force the regeneration of the grammar parser. If the parser is out of date it will be regenerated and subsequently built.
<code>--with-python=path</code>	Enables Python integration, given a path to the Python libraries.
<code>--gc</code>	Enables use of the Boehm Garbage Collector. The build will look for the Boehm-GC source in a "boehm_gc" subdirectory from the <code>bjam</code> sources.

<code>--duma</code>	Enables use of the DUMA (Detect Uintended Memory Access) debugging memory allocator. The build expects to find the DUMA source files in a "duma" subdirectory from the <code>bjam</code> sources.
<code>--toolset-root=path</code>	Indicates where the toolset used to build is located. This option is passed in by the bootstrap (<code>build.bat</code> or <code>build.sh</code>) script.
<code>--show-locate-target</code>	For information, prints out where it will put the built executable.
<code>--noassert</code>	Disable debug assertions, even if building the debug version of the executable.
<code>dist</code>	Generate packages (compressed archives) as appropriate for distribution in the platform, if possible.
<code>clean</code>	Remove all the built executables and objects.

Using BJam



Warning

Most probably, you are looking for [Boost.Build manual](#) or [Boost.Build command-line syntax](#). This section documents only low-level options used by the Boost.Jam build engine, and does not mention any high-level syntax of Boost.Build

If *target* is provided on the command line, `bjam` builds *target*; otherwise `bjam` builds the target `all`.

```
bjam ( -option [value] | target ) *
```

Options

Options are either singular or have an accompanying value. When a value is allowed, or required, it can be either given as an argument following the option argument, or it can be given immediately after the option as part of the option argument. The allowed options are:

- | | |
|-------------------|--|
| <code>-a</code> | Build all targets anyway, even if they are up-to-date. |
| <code>-d n</code> | Enable cumulative debugging levels from 1 to n. Values are: <ol style="list-style-type: none">1. Show the actions taken for building targets, as they are executed (the default).2. Show "quiet" actions and display all action text, as they are executed.3. Show dependency analysis, and target/source timestamps/paths.4. Show arguments and timing of shell invocations.5. Show rule invocations and variable expansions.6. Show directory/header file/archive scans, and attempts at binding to targets.7. Show variable settings.8. Show variable fetches, variable expansions, and evaluation of "if" expressions.9. Show variable manipulation, scanner tokens, and memory usage. |

	10. Show profile information for rules, both timing and memory.
	11. Show parsing progress of Jamfiles.
	12. Show graph of target dependencies.
	13. Show change target status (fate).
<code>-d +n</code>	Enable debugging level <i>n</i> .
<code>-d 0</code>	Turn off all debugging levels. Only errors are reported.
<code>-f <i>Jambase</i></code>	Read <i>Jambase</i> instead of using the built-in Jambase. Only one <code>-f</code> flag is permitted, but the <i>Jambase</i> may explicitly include other files. A <i>Jambase</i> name of "-" is allowed, in which case console input is read until it is closed, at which point the input is treated as the Jambase.
<code>-j n</code>	Run up to <i>n</i> shell commands concurrently (UNIX and NT only). The default is 1.
<code>-l n</code>	Limit actions to running for <i>n</i> number of seconds, after which they are stopped. Note: Windows only.
<code>-n</code>	Don't actually execute the updating actions, but do everything else. This changes the debug level default to <code>-d 2</code> .
<code>-o <i>file</i></code>	Write the updating actions to the specified file instead of running them.
<code>-q</code>	Quit quickly (as if an interrupt was received) as soon as any target fails.
<code>-s <i>var=value</i></code>	Set the variable <i>var</i> to <i>value</i> , overriding both internal variables and variables imported from the environment.
<code>-t <i>target</i></code>	Rebuild <i>target</i> and everything that depends on it, even if it is up-to-date.
<code>-- <i>value</i></code>	The option and <i>value</i> is ignored, but is available from the <code>\$ (ARGV)</code> variable.
<code>-v</code>	Print the version of bjam and exit.

Command-line and Environment Variable Quoting

Classic Jam had an odd behavior with respect to command-line variable (`-s . . .`) and environment variable settings which made it impossible to define an arbitrary variable with spaces in the value. Boost Jam remedies that by treating all such settings as a single string if they are surrounded by double-quotes. Uses of this feature can look interesting, since shells require quotes to keep characters separated by whitespace from being treated as separate arguments:

```
jam -sMSVCNT="\\"C:\Program Files\Microsoft Visual C++\VC98\\" ...
```

The outer quote is for the shell. The middle quote is for Jam, to tell it to take everything within those quotes literally, and the inner quotes are for the shell again when paths are passed as arguments to build actions. Under NT, it looks a lot more sane to use environment variables before invoking jam when you have to do this sort of quoting:

```
set MSVCNT=""C:\Program Files\Microsoft Visual C++\VC98\""
```

Operation

BJam has four phases of operation: start-up, parsing, binding, and updating.

Start-up

Upon start-up, bjam imports environment variable settings into bjam variables. Environment variables are split at blanks with each word becoming an element in the variable's list of values. Environment variables whose names end in `PATH` are split at `$(SPLITPATH)` characters (e.g., `:` for Unix).

To set a variable's value on the command line, overriding the variable's environment value, use the `-s` option. To see variable assignments made during bjam's execution, use the `-d+7` option.

The Boost.Build v2 initialization behavior has been implemented. This behavior only applies when the executable being invoked is called "bjam" or, for backward-compatibility, when the `BOOST_ROOT` variable is set.

1. We attempt to load "boost-build.jam" by searching from the current invocation directory up to the root of the file system. This file is expected to invoke the `boost-build` rule to indicate where the Boost.Build system files are, and to load them.
2. If `boost-build.jam` is not found we error and exit, giving brief instructions on possible errors. As a backward-compatibility measure for older versions of Boost.Build, when the `BOOST_ROOT` variable is set, we first search for `boost-build.jam` in `$(BOOST_ROOT)/tools/build` and `$(BOOST_BUILD_PATH)`. If found, it is loaded and initialization is complete.
3. The `boost-build` rule adds its (optional) argument to the front of `BOOST_BUILD_PATH`, and attempts to load `bootstrap.jam` from those directories. If a relative path is specified as an argument, it is treated as though it was relative to the `boost-build.jam` file.
4. If the `bootstrap.jam` file was not found, we print a likely error message and exit.

Parsing

In the parsing phase, bjam reads and parses the Jambase file, by default the built-in one. It is written in the [jam language](#). The last action of the Jambase is to read (via the "include" rule) a user-provided file called "Jamfile".

Collectively, the purpose of the Jambase and the Jamfile is to name build targets and source files, construct the dependency graph among them, and associate build actions with targets. The Jambase defines boilerplate rules and variable assignments, and the Jamfile uses these to specify the actual relationship among the target and source files.

Binding

After parsing, bjam recursively descends the dependency graph and binds every file target with a location in the filesystem. If bjam detects a circular dependency in the graph, it issues a warning.

File target names are given as absolute or relative path names in the filesystem. If the path name is absolute, it is bound as is. If the path name is relative, it is normally bound as is, and thus relative to the current directory. This can be modified by the settings of the `$(SEARCH)` and `$(LOCATE)` variables, which enable jam to find and build targets spread across a directory tree. See [SEARCH and LOCATE Variables](#) below.

Update Determination

After binding each target, bjam determines whether the target needs updating, and if so marks the target for the updating phase. A target is normally so marked if it is missing, it is older than any of its sources, or any of its sources are marked for updating. This behavior can be modified by the application of special built-in rules, `ALWAYS`, `LEAVES`, `NO CARE`, `NOT FILE`, `NO UPDATE`, and `TEMPORARY`. See [Modifying Binding](#) below.

Header File Scanning

During the binding phase, `bjam` also performs header file scanning, where it looks inside source files for the implicit dependencies on other files caused by C's `#include` syntax. This is controlled by the special variables `$(HDRSCAN)` and `$(HRRULE)`. The result of the scan is formed into a rule invocation, with the scanned file as the target and the found included file names as the sources. Note that this is the only case where rules are invoked outside the parsing phase. See [HDRSCAN and HRRULE Variables](#) below.

Updating

After binding, `bjam` again recursively descends the dependency graph, this time executing the update actions for each target marked for update during the binding phase. If a target's updating actions fail, then all other targets which depend on that target are skipped.

The `-j` flag instructs `bjam` to build more than one target at a time. If there are multiple actions on a single target, they are run sequentially.

Language

`BJam` has an interpreted, procedural language. Statements in `bjam` are rule (procedure) definitions, rule invocations, flow-of-control structures, variable assignments, and sundry language support.

Lexical Features

`BJam` treats its input files as whitespace-separated tokens, with two exceptions: double quotes (") can enclose whitespace to embed it into a token, and everything between the matching curly braces ({}) in the definition of a rule action is treated as a single string. A backslash (\) can escape a double quote, or any single whitespace character.

`BJam` requires whitespace (blanks, tabs, or newlines) to surround all tokens, including the colon (:) and semicolon (;) tokens.

`BJam` keywords (as mentioned in this document) are reserved and generally must be quoted with double quotes (") to be used as arbitrary tokens, such as variable or target names.

Comments start with the `#` character and extend until the end of line.

Targets

The essential `bjam` data entity is a target. Build targets are files to be updated. Source targets are the files used in updating built targets. Built targets and source targets are collectively referred to as file targets, and frequently built targets are source targets for other built targets. Pseudotargets are symbols which represent dependencies on other targets, but which are not themselves associated with any real file.

A file target's identifier is generally the file's name, which can be absolutely rooted, relative to the directory of `bjam`'s invocation, or simply local (no directory). Most often it is the last case, and the actual file path is bound using the `$(SEARCH)` and `$(LOCATE)` special variables. See [SEARCH and LOCATE Variables](#) below. A local filename is optionally qualified with `grist`, a string value used to assure uniqueness. A file target with an identifier of the form `file(member)` is a library member (usually an `ar(1)` archive on Unix).

Binding Detection

Whenever a target is bound to a location in the filesystem, Boost Jam will look for a variable called `BINDRULE` (first "on" the target being bound, then in the global module). If non-empty, `=$(BINDRULE[1])=` names a rule which is called with the name of the target and the path it is being bound to. The signature of the rule named by `=$(BINDRULE[1])=` should match the following:

```
rule bind-rule ( target : path )
```

This facility is useful for correct header file scanning, since many compilers will search for `#include` files first in the directory containing the file doing the `#include` directive. `$(BINDRULE)` can be used to make a record of that directory.

Rules

The basic `bjam` language entity is called a rule. A rule is defined in two parts: the procedure and the actions. The procedure is a body of jam statements to be run when the rule is invoked; the actions are the OS shell commands to execute when updating the built targets of the rule.

Rules can return values, which can be expanded into a list with "[*rule args ...*]". A rule's value is the value of its last statement, though only the following statements have values: 'if' (value of the leg chosen), 'switch' (value of the case chosen), set (value of the resulting variable), and 'return' (value of its arguments). Note that 'return' doesn't actually cause a return, i.e., is a no-op unless it is the last statement of the last block executed within rule body.

The `bjam` statements for defining and invoking rules are as follows:

Define a rule's procedure, replacing any previous definition.

```
rule rulename { statements }
```

Define a rule's updating actions, replacing any previous definition.

```
actions [ modifiers ] rulename { commands }
```

Invoke a rule.

```
rulename field1 : field2 : ... : fieldN ;
```

Invoke a rule under the influence of target's specific variables..

```
on target rulename field1 : field2 : ... : fieldN ;
```

Used as an argument, expands to the return value of the rule invoked.

```
[ rulename field1 : field2 : ... : fieldN ]  
[ on target rulename field1 : field2 : ... : fieldN ]
```

A rule is invoked with values in *field1* through *fieldN*. They may be referenced in the procedure's statements as $\$(1)$ through $\$(N)$ (9 max), and the first two only may be referenced in the action's *commands* as $\$(1)$ and $\$(2)$. $\$(<)$ and $\$(>)$ are synonymous with $\$(1)$ and $\$(2)$.

Rules fall into two categories: updating rules (with actions), and pure procedure rules (without actions). Updating rules treat arguments $\$(1)$ and $\$(2)$ as built targets and sources, respectively, while pure procedure rules can take arbitrary arguments.

When an updating rule is invoked, its updating actions are added to those associated with its built targets ($\$(1)$) before the rule's procedure is run. Later, to build the targets in the updating phase, *commands* are passed to the OS command shell, with $\$(1)$ and $\$(2)$ replaced by bound versions of the target names. See Binding above.

Rule invocation may be indirected through a variable:

```
$(var) field1 : field2 : ... : fieldN ;

on target $(var) field1 : field2 : ... : fieldN ;

[ $(var) field1 : field2 : ... : fieldN ]
[ on target $(var) field1 : field2 : ... : fieldN ]
```

The variable's value names the rule (or rules) to be invoked. A rule is invoked for each element in the list of `$(var)`'s values. The fields `field1 : field2 : ...` are passed as arguments for each invocation. For the `[...]` forms, the return value is the concatenation of the return values for all of the invocations.

Action Modifiers

The following action modifiers are understood:

<code>actions bind vars</code>	<code>\$(vars)</code> will be replaced with bound values.
<code>actions existing</code>	<code>\$(>)</code> includes only source targets currently existing.
<code>actions ignore</code>	The return status of the commands is ignored.
<code>actions piecemeal</code>	commands are repeatedly invoked with a subset of <code>\$(>)</code> small enough to fit in the command buffer on this OS.
<code>actions quietly</code>	The action is not echoed to the standard output.
<code>actions together</code>	The <code>\$(>)</code> from multiple invocations of the same action on the same built target are glommed together.
<code>actions updated</code>	<code>\$(>)</code> includes only source targets themselves marked for updating.

Argument lists

You can describe the arguments accepted by a rule, and refer to them by name within the rule. For example, the following prints "I'm sorry, Dave" to the console:

```
rule report ( pronoun index ? : state : names + )
{
    local he.suffix she.suffix it.suffix = s ;
    local I.suffix = m ;
    local they.suffix you.suffix = re ;
    ECHO $(pronoun)'$(pronoun).suffix' $(state), $(names[$(index)]) ;
}
report I 2 : sorry : Joe Dave Pete ;
```

Each name in a list of formal arguments (separated by ":" in the rule declaration) is bound to a single element of the corresponding actual argument unless followed by one of these modifiers:

Symbol	Semantics of preceding symbol
?	optional
*	Bind to zero or more unbound elements of the actual argument. When * appears where an argument name is expected, any number of additional arguments are accepted. This feature can be used to implement "varargs" rules.
+	Bind to one or more unbound elements of the actual argument.

The actual and formal arguments are checked for inconsistencies, which cause Jam to exit with an error code:

```

### argument error
# rule report ( pronoun index ? : state : names + )
# called with: ( I 2 foo : sorry : Joe Dave Pete )
# extra argument foo
### argument error
# rule report ( pronoun index ? : state : names + )
# called with: ( I 2 : sorry )
# missing argument names

```

If you omit the list of formal arguments, all checking is bypassed as in "classic" Jam. Argument lists drastically improve the reliability and readability of your rules, however, and are **strongly recommended** for any new Jam code you write.

Built-in Rules

BJam has a growing set of built-in rules, all of which are pure procedure rules without updating actions. They are in three groups: the first builds the dependency graph; the second modifies it; and the third are just utility rules.

Dependency Building

DEPENDS

```
rule DEPENDS ( targets1 * : targets2 * )
```

Builds a direct dependency: makes each of *targets1* depend on each of *targets2*. Generally, *targets1* will be rebuilt if *targets2* are themselves rebuilt or are newer than *targets1*.

INCLUDES

```
rule INCLUDES ( targets1 * : targets2 * )
```

Builds a sibling dependency: makes any target that depends on any of *targets1* also depend on each of *targets2*. This reflects the dependencies that arise when one source file includes another: the object built from the source file depends both on the original and included source file, but the two sources files don't depend on each other. For example:

```
DEPENDS foo.o : foo.c ;
INCLUDES foo.c : foo.h ;
```

"foo.o" depends on "foo.c" and "foo.h" in this example.

Modifying Binding

The six rules ALWAYS, LEAVES, NOCARE, NOTFILE, NOUPDATE, and TEMPORARY modify the dependency graph so that bjam treats the targets differently during its target binding phase. See Binding above. Normally, bjam updates a target if it is missing, if its filesystem modification time is older than any of its dependencies (recursively), or if any of its dependencies are being updated. This basic behavior can be changed by invoking the following rules:

ALWAYS

```
rule ALWAYS ( targets * )
```

Causes *targets* to be rebuilt regardless of whether they are up-to-date (they must still be in the dependency graph). This is used for the clean and uninstall targets, as they have no dependencies and would otherwise appear never to need building. It is best applied to targets that are also NOTFILE targets, but it can also be used to force a real file to be updated as well.

LEAVES

```
rule LEAVES ( targets * )
```

Makes each of *targets* depend only on its leaf sources, and not on any intermediate targets. This makes it immune to its dependencies being updated, as the "leaf" dependencies are those without their own dependencies and without updating actions. This allows a target to be updated only if original source files change.

NOCARE

```
rule NOCARE ( targets * )
```

Causes *bjam* to ignore *targets* that neither can be found nor have updating actions to build them. Normally for such targets *bjam* issues a warning and then skips other targets that depend on these missing targets. The `HdrRule` in *Jambase* uses `NOCARE` on the header file names found during header file scanning, to let *bjam* know that the included files may not exist. For example, if an `#include` is within an `#ifdef`, the included file may not actually be around.

**Warning**

For targets with build actions: if their build actions exit with a nonzero return code, dependent targets will still be built.

NOTFILE

```
rule NOTFILE ( targets * )
```

Marks *targets* as pseudotargets and not real files. No timestamp is checked, and so the actions on such a target are only executed if the target's dependencies are updated, or if the target is also marked with `ALWAYS`. The default *bjam* target "all" is a pseudotarget. In *Jambase*, `NOTFILE` is used to define several additional convenient pseudotargets.

NOUPDATE

```
rule NOUPDATE ( targets * )
```

Causes the timestamps on *targets* to be ignored. This has two effects: first, once the target has been created it will never be updated; second, manually updating target will not cause other targets to be updated. In *Jambase*, for example, this rule is applied to directories by the `MkDir` rule, because `MkDir` only cares that the target directory exists, not when it has last been updated.

TEMPORARY

```
rule TEMPORARY ( targets * )
```

Marks *targets* as temporary, allowing them to be removed after other targets that depend upon them have been updated. If a `TEMPORARY` target is missing, *bjam* uses the timestamp of the target's parent. *Jambase* uses `TEMPORARY` to mark object files that are archived in a library after they are built, so that they can be deleted after they are archived.

FAIL_EXPECTED

```
rule FAIL_EXPECTED ( targets * )
```

For handling targets whose build actions are expected to fail (e.g. when testing that assertions or compile-time type checking work properly), Boost Jam supplies the `FAIL_EXPECTED` rule in the same style as `NOCARE`, et. al. During target updating, the return code of the build actions for arguments to `FAIL_EXPECTED` is inverted: if it fails, building of dependent targets continues as though it succeeded. If it succeeds, dependent targets are skipped.

RMOLD

```
rule RMOLD ( targets * )
```

BJam removes any target files that may exist on disk when the rule used to build those targets fails. However, targets whose dependencies fail to build are not removed by default. The `RMOLD` rule causes its arguments to be removed if any of their dependencies fail to build.

ISFILE

```
rule ISFILE ( targets * )
```

`ISFILE` marks targets as required to be files. This changes the way bjam searches for the target such that it ignores mathes for file system items that are not file, like directories. This makes it possible to avoid `#include "exception"` matching if one happens to have a directory named `exception` in the header search path.



Warning

This is currently not fully implemented.

Utility

The two rules `ECHO` and `EXIT` are utility rules, used only in bjam's parsing phase.

ECHO

```
rule ECHO ( args * )
```

Blurts out the message *args* to stdout.

EXIT

```
rule EXIT ( message * : result-value ? )
```

Blurts out the *message* to stdout and then exits with a failure status if no *result-value* is given, otherwise it exits with the given *result-value*.

"Echo", "echo", "Exit", and "exit" are accepted as aliases for `ECHO` and `EXIT`, since it is hard to tell that these are built-in rules and not part of the language, like "include".

GLOB

The `GLOB` rule does filename globbing.

```
rule GLOB ( directories * : patterns * : downcase-opt ? )
```

Using the same wildcards as for the patterns in the switch statement. It is invoked by being used as an argument to a rule invocation inside of "[]". For example: `FILES = [GLOB dir1 dir2 : *.c *.h]` sets `FILES` to the list of C source and header files in `dir1` and `dir2`. The resulting filenames are the full pathnames, including the directory, but the pattern is applied only to the file name without the directory.

If *downcase-opt* is supplied, filenames are converted to all-lowercase before matching against the pattern; you can use this to do case-insensitive matching using lowercase patterns. The paths returned will still have mixed case if the OS supplies them. On Windows NT and Cygwin, filenames are always downcased before matching.

MATCH

The MATCH rule does pattern matching.

```
rule MATCH ( regexps + : list * )
```

Matches the `egrep(1)` style regular expressions *regexps* against the strings in *list*. The result is the concatenation of matching () subexpressions for each string in *list*, and for each regular expression in *regexps*. Only useful within the "[]" construct, to change the result into a list.

BACKTRACE

```
rule BACKTRACE ( )
```

Returns a list of quadruples: *filename line module rulename...*, describing each shallower level of the call stack. This rule can be used to generate useful diagnostic messages from Jam rules.

UPDATE

```
rule UPDATE ( targets * )
```

Classic jam treats any non-option element of command line as a name of target to be updated. This prevented more sophisticated handling of command line. This is now enabled again but with additional changes to the UPDATE rule to allow for the flexibility of changing the list of targets to update. The UPDATE rule has two effects:

1. It clears the list of targets to update, and
2. Causes the specified targets to be updated.

If no target was specified with the UPDATE rule, no targets will be updated. To support changing of the update list in more useful ways, the rule also returns the targets previously in the update list. This makes it possible to add targets as such:

```
local previous-updates = [ UPDATE ] ;  
UPDATE $(previous-updates) a-new-target ;
```

W32_GETREG

```
rule W32_GETREG ( path : data ? )
```

Defined only for win32 platform. It reads the registry of Windows. '*path*' is the location of the information, and '*data*' is the name of the value which we want to get. If '*data*' is omitted, the default value of '*path*' will be returned. The '*path*' value must conform to MS key path format and must be prefixed with one of the predefined root keys. As usual,

- 'HKLM' is equivalent to 'HKEY_LOCAL_MACHINE'.
- 'HKCU' is equivalent to 'HKEY_CURRENT_USER'.
- 'HKCR' is equivalent to 'HKEY_CLASSES_ROOT'.

Other predefined root keys are not supported.

Currently supported data types : 'REG_DWORD', 'REG_SZ', 'REG_EXPAND_SZ', 'REG_MULTI_SZ'. The data with 'REG_DWORD' type will be turned into a string, 'REG_MULTI_SZ' into a list of strings, and for those with 'REG_EXPAND_SZ' type environment variables in it will be replaced with their defined values. The data with 'REG_SZ' type and other unsupported types will be put into a string without modification. If it can't receive the value of the data, it just return an empty list. For example,

```
local PSDK-location =
  [ W32_GETREG HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\MicrosoftSDK\\Directories : "Install ↵
Dir" ] ;
```

W32_GETREGNAMES

```
rule W32_GETREGNAMES ( path : result-type )
```

Defined only for win32 platform. It reads the registry of Windows. '*path*' is the location of the information, and '*result-type*' is either 'subkeys' or 'values'. For more information on '*path*' format and constraints, please see W32_GETREG.

Depending on '*result-type*', the rule returns one of the following:

- subkeys** Names of all direct subkeys of '*path*'.
- values** Names of values contained in registry key given by '*path*'. The "default" value of the key appears in the returned list only if its value has been set in the registry.

If '*result-type*' is not recognized, or requested data cannot be retrieved, the rule returns an empty list. Example:

```
local key = "HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\App Paths" ;
local subkeys = [ W32_GETREGNAMES "${key}" : subkeys ] ;
for local subkey in $(subkeys)
{
  local values = [ W32_GETREGNAMES "${key}\\$(subkey)" : values ] ;
  for local value in $(values)
  {
    local data = [ W32_GETREG "${key}\\$(subkey)" : "${value}" ] ;
    ECHO "Registry path: " $(key)\\$(subkey) ":" $(value) "=" $(data) ;
  }
}
```

SHELL

```
rule SHELL ( command : * )
```

SHELL executes *command*, and then returns the standard output of *command*. SHELL only works on platforms with a `popen()` function in the C library. On platforms without a working `popen()` function, SHELL is implemented as a no-op. SHELL works on Unix, MacOS X, and most Windows compilers. SHELL is a no-op on Metrowerks compilers under Windows. There is a variable set of allowed options as additional arguments:

- exit-status** In addition to the output the result status of the executed command is returned as a second element of the result.
- no-output** Don't capture the output of the command. Instead an empty ("") string value is returned in place of the output.

Because the Perforce/Jambase defines a SHELL rule which hides the builtin rule, COMMAND can be used as an alias for SHELL in such a case.

MD5

```
rule MD5 ( string )
```

MD5 computes the MD5 hash of the string passed as parameter and returns it.

SPLIT_BY_CHARACTERS

```
rule SPLIT_BY_CHARACTERS ( string : delimiters )
```

SPLIT_BY_CHARACTERS splits the specified *string* on any delimiter character present in *delimiters* and returns the resulting list.

PRECIOUS

```
rule PRECIOUS ( targets * )
```

The PRECIOUS rule specifies that each of the targets passed as the arguments should not be removed even if the command updating that target fails.

PAD

```
rule PAD ( string : width )
```

If *string* is shorter than *width* characters, pads it with whitespace characters on the right, and returns the result. Otherwise, returns *string* unmodified.

FILE_OPEN

```
rule FILE_OPEN ( filename : mode )
```

The FILE_OPEN rule opens the specified file and returns a file descriptor. The *mode* parameter can be either "w" or "r". Note that at present, only the UPDATE_NOW rule can use the resulting file descriptor number.

UPDATE_NOW

```
rule UPDATE_NOW ( targets * : log ? : ignore-minus-n ? )
```

The UPDATE_NOW caused the specified targets to be updated immediately. If update was successful, non-empty string is returned. The *log* parameter, if present, specifies a descriptor of a file where all output from building is redirected. If the *ignore-minus-n* parameter is specified, the targets are updated even if the *-n* parameter is specified on the command line.

Flow-of-Control

BJam has several simple flow-of-control statements:

```
for var in list { statements }
```

Executes *statements* for each element in *list*, setting the variable *var* to the element value.

```
if cond { statements }  
[ else { statements } ]
```

Does the obvious; the *else* clause is optional. *cond* is built of:

- | | |
|----------------------|---|
| <i>a</i> | true if any <i>a</i> element is a non-zero-length string |
| <i>a</i> = <i>b</i> | list <i>a</i> matches list <i>b</i> string-for-string |
| <i>a</i> != <i>b</i> | list <i>a</i> does not match list <i>b</i> |
| <i>a</i> < <i>b</i> | <i>a</i> [<i>i</i>] string is less than <i>b</i> [<i>i</i>] string, where <i>i</i> is first mismatched element in lists <i>a</i> and <i>b</i> |

<code>a <= b</code>	every <i>a</i> string is less than or equal to its <i>b</i> counterpart
<code>a > b</code>	<i>a</i> [<i>i</i>] string is greater than <i>b</i> [<i>i</i>] string, where <i>i</i> is first mismatched element
<code>a >= b</code>	every <i>a</i> string is greater than or equal to its <i>b</i> counterpart
<code>a in b</code>	true if all elements of <i>a</i> can be found in <i>b</i> , or if <i>a</i> has no elements
<code>! cond</code>	condition not true
<code>cond && cond</code>	conjunction
<code>cond cond</code>	disjunction
<code>(cond)</code>	precedence grouping

```
include file ;
```

Causes `bjam` to read the named *file*. The *file* is bound like a regular target (see Binding above) but unlike a regular target the include *file* cannot be built.

The include *file* is inserted into the input stream during the parsing phase. The primary input file and all the included file(s) are treated as a single file; that is, jam infers no scope boundaries from included files.

```
local vars [ = values ] ;
```

Creates new *vars* inside to the enclosing `{ }` block, obscuring any previous values they might have. The previous values for *vars* are restored when the current block ends. Any rule called or file included will see the local and not the previous value (this is sometimes called Dynamic Scoping). The local statement may appear anywhere, even outside of a block (in which case the previous value is restored when the input ends). The *vars* are initialized to *values* if present, or left uninitialized otherwise.

```
return values ;
```

Within a rule body, the return statement sets the return value for an invocation of the rule. It does **not** cause the rule to return; a rule's value is actually the value of the last statement executed, so a return should be the last statement executed before the rule "naturally" returns.

```
switch value
{
    case pattern1 : statements ;
    case pattern2 : statements ;
    ...
}
```

The switch statement executes zero or one of the enclosed *statements*, depending on which, if any, is the first case whose *pattern* matches *value*. The *pattern* values are not variable-expanded. The pattern values may include the following wildcards:

<code>?</code>	match any single character
<code>*</code>	match zero or more characters
<code>[chars]</code>	match any single character in <i>chars</i>
<code>[^chars]</code>	match any single character not in <i>chars</i>
<code>\x</code>	match <i>x</i> (escapes the other wildcards)

```
while cond { statements }
```

Repeatedly execute *statements* while *cond* remains true upon entry. (See the description of *cond* expression syntax under if, above).

Variables

BJam variables are lists of zero or more elements, with each element being a string value. An undefined variable is indistinguishable from a variable with an empty list, however, a defined variable may have one more elements which are null strings. All variables are referenced as `$(variable)`.

Variables are either global or target-specific. In the latter case, the variable takes on the given value only during the updating of the specific target.

A variable is defined with:

```
variable = elements ;  
variable += elements ;  
variable on targets = elements ;  
variable on targets += elements ;  
variable default = elements ;  
variable ?= elements ;
```

The first two forms set *variable* globally. The third and forth forms set a target-specific variable. The = operator replaces any previous elements of *variable* with *elements*; the += operation adds *elements* to *variable*'s list of elements. The final two forms are synonymous: they set *variable* globally, but only if it was previously unset.

Variables referenced in updating commands will be replaced with their values; target-specific values take precedence over global values. Variables passed as arguments (`$(1)` and `$(2)`) to actions are replaced with their bound values; the "bind" modifier can be used on actions to cause other variables to be replaced with bound values. See Action Modifiers above.

BJam variables are not re-exported to the environment of the shell that executes the updating actions, but the updating actions can reference bjam variables with `$(variable)`.

Variable Expansion

During parsing, bjam performs variable expansion on each token that is not a keyword or rule name. Such tokens with embedded variable references are replaced with zero or more tokens. Variable references are of the form `$(v)` or `$(vm)`, where *v* is the variable name, and *m* are optional modifiers.

Variable expansion in a rule's actions is similar to variable expansion in statements, except that the action string is tokenized at whitespace regardless of quoting.

The result of a token after variable expansion is the *product* of the components of the token, where each component is a literal substring or a list substituting a variable reference. For example:

```
$(X) -> a b c  
t$(X) -> ta tb tc  
$(X)z -> az bz cz  
$(X)-$(X) -> a-a a-b a-c b-a b-b b-c c-a c-b c-c
```

The variable name and modifiers can themselves contain a variable reference, and this partakes of the product as well:

```
$(X) -> a b c
$(Y) -> 1 2
$(Z) -> X Y
$($(Z)) -> a b c 1 2
```

Because of this product expansion, if any variable reference in a token is undefined, the result of the expansion is an empty list. If any variable element is a null string, the result propagates the non-null elements:

```
$(X) -> a ""
$(Y) -> "" 1
$(Z) ->
-$(X)$(Y)- -> -a- -a1- -- -1-
-$(X)$(Z)- ->
```

A variable element's string value can be parsed into grist and filename-related components. Modifiers to a variable are used to select elements, select components, and replace components. The modifiers are:

- [*n*] Select element number *n* (starting at 1). If the variable contains fewer than *n* elements, the result is a zero-element list. *n* can be negative in which case the element number *n* from the last leftward is returned.
- [*n-m*] Select elements number *n* through *m*. *n* and *m* can be negative in which case they refer to elements counting from the last leftward.
- [*n-*] Select elements number *n* through the last. *n* can be negative in which case it refers to the element counting from the last leftward.
- :B Select filename base.
- :S Select (last) filename suffix.
- :M Select archive member name.
- :D Select directory path.
- :P Select parent directory.
- :G Select grist.
- :U Replace lowercase characters with uppercase.
- :L Replace uppercase characters with lowercase.
- :T Converts all back-slashes ("\") to forward slashes ("/"). For example

```
x = "C:\\Program Files\\Borland" ; ECHO ${x:T} ;
```

```
prints "C:/Program Files/Borland"
```

- :W When invoking Windows-based tools from [Cygwin](#) it can be important to pass them true windows-style paths. The :w modifier, **under Cygwin only**, turns a cygwin path into a Win32 path using the [cygwin_conv_to_win32_path](#) function. On other platforms, the string is unchanged. For example

```
x = "/cygdrive/c/Program Files/Borland" ; ECHO ${x:W} ;
```

```
prints "C:\\Program Files\\Borland" on Cygwin
```

- :*chars* Select the components listed in *chars*.

<code>:G=grist</code>	Replace grist with <i>grist</i> .
<code>:D=path</code>	Replace directory with <i>path</i> .
<code>:B=base</code>	Replace the base part of file name with <i>base</i> .
<code>:S=suf</code>	Replace the suffix of file name with <i>suf</i> .
<code>:M=mem</code>	Replace the archive member name with <i>mem</i> .
<code>:R=root</code>	Prepend <i>root</i> to the whole file name, if not already rooted.
<code>:E=value</code>	Assign <i>value</i> to the variable if it is unset.
<code>:J=joinval</code>	Concatenate list elements into single element, separated by <i>joinval</i> .

On VMS, `$(var:P)` is the parent directory of `$(var:D)`.

Local For Loop Variables

Boost Jam allows you to declare a local for loop control variable right in the loop:

```
x = 1 2 3 ;
y = 4 5 6 ;
for local y in $(x)
{
    ECHO $(y) ; # prints "1", "2", or "3"
}
ECHO $(y) ;    # prints "4 5 6"
```

Generated File Expansion

During expansion of expressions `bjam` also looks for subexpressions of the form `=@(filename:Efilecontents)` and replaces the expression with `filename` after creating the given file with the contents set to `filecontents`. This is useful for creating compiler response files, and other "internal" files. The expansion works both during parsing and action execution. Hence it is possible to create files during any of the three build phases.

Built-in Variables

This section discusses variables that have special meaning to `bjam`. All of these must be defined or used in the global module -- using those variables inside a named module will not have the desired effect. See [Modules](#).

SEARCH and LOCATE

These two variables control the binding of file target names to locations in the file system. Generally, `$(SEARCH)` is used to find existing sources while `$(LOCATE)` is used to fix the location for built targets.

Rooted (absolute path) file targets are bound as is. Unrooted file target names are also normally bound as is, and thus relative to the current directory, but the settings of `$(LOCATE)` and `$(SEARCH)` alter this:

- If `$(LOCATE)` is set then the target is bound relative to the first directory in `$(LOCATE)`. Only the first element is used for binding.
- If `$(SEARCH)` is set then the target is bound to the first directory in `$(SEARCH)` where the target file already exists.
- If the `$(SEARCH)` search fails, the target is bound relative to the current directory anyhow.

Both `$(SEARCH)` and `$(LOCATE)` should be set target-specific and not globally. If they were set globally, `bjam` would use the same paths for all file binding, which is not likely to produce sane results. When writing your own rules, especially ones not built upon those in Jambase, you may need to set `$(SEARCH)` or `$(LOCATE)` directly. Almost all of the rules defined in Jambase set `$(SEARCH)` and `$(LOCATE)` to sensible values for sources they are looking for and targets they create, respectively.

HDRSCAN and HDRRULE

These two variables control header file scanning. `$(HDRSCAN)` is an `egrep(1)` pattern, with `()`'s surrounding the file name, used to find file inclusion statements in source files. `Jambase` uses `$(HDRPATTERN)` as the pattern for `$(HDRSCAN)`. `$(HDRRULE)` is the name of a rule to invoke with the results of the scan: the scanned file is the target, the found files are the sources. This is the only place where `bjam` invokes a rule through a variable setting.

Both `$(HDRSCAN)` and `$(HDRRULE)` must be set for header file scanning to take place, and they should be set target-specific and not globally. If they were set globally, all files, including executables and libraries, would be scanned for header file include statements.

The scanning for header file inclusions is not exact, but it is at least dynamic, so there is no need to run something like `make-depend(GNU)` to create a static dependency file. The scanning mechanism errs on the side of inclusion (i.e., it is more likely to return filenames that are not actually used by the compiler than to miss include files) because it can't tell if `#include` lines are inside `#ifdefs` or other conditional logic. In `Jambase`, `HdrRule` applies the `NOCARE` rule to each header file found during scanning so that if the file isn't present yet doesn't cause the compilation to fail, `bjam` won't care.

Also, scanning for regular expressions only works where the included file name is literally in the source file. It can't handle languages that allow including files using variable names (as the `Jam` language itself does).

Semaphores

It is sometimes desirable to disallow parallel execution of some actions. For example:

- Old versions of `yacc` use files with fixed names. So, running two `yacc` actions is dangerous.
- One might want to perform parallel compiling, but not do parallel linking, because linking is i/o bound and only gets slower.

Craig McPeeters has extended `Perforce Jam` to solve such problems, and that extension was integrated in `Boost.Jam`.

Any target can be assigned a *semaphore*, by setting a variable called `SEMAPHORE` on that target. The value of the variable is the semaphore name. It must be different from names of any declared target, but is arbitrary otherwise.

The semantic of semaphores is that in a group of targets which have the same semaphore, only one can be updated at the moment, regardless of `"-j"` option.

Platform Identifier

A number of `Jam` built-in variables can be used to identify runtime platform:

<code>OS</code>	OS identifier string
<code>OSPLAT</code>	Underlying architecture, when applicable
<code>MAC</code>	true on MAC platform
<code>NT</code>	true on NT platform
<code>OS2</code>	true on OS2 platform
<code>UNIX</code>	true on Unix platforms
<code>VMS</code>	true on VMS platform

Jam Version

<code>JAMDATE</code>	Time and date at <code>bjam</code> start-up as an ISO-8601 UTC value.
<code>JAMUNAME</code>	Output of <code>uname(1)</code> command (Unix only)
<code>JAMVERSION</code>	<code>bjam</code> version, currently "3.1.18"

`JAM_VERSION` A predefined global variable with two elements indicates the version number of Boost Jam. Boost Jam versions start at "03" "00". Earlier versions of Jam do not automatically define `JAM_VERSION`.

JAMSHELL

When `bjam` executes a rule's action block, it forks and execs a shell, passing the action block as an argument to the shell. The invocation of the shell can be controlled by `$(JAMSHELL)`. The default on Unix is, for example:

```
JAMSHELL = /bin/sh -c % ;
```

The `%` is replaced with the text of the action block.

`BJam` does not directly support building in parallel across multiple hosts, since that is heavily dependent on the local environment. To build in parallel across multiple hosts, you need to write your own shell that provides access to the multiple hosts. You then reset `$(JAMSHELL)` to reference it.

Just as `bjam` expands a `%` to be the text of the rule's action block, it expands a `!` to be the multi-process slot number. The slot number varies between 1 and the number of concurrent jobs permitted by the `-j` flag given on the command line. Armed with this, it is possible to write a multiple host shell. For example:

```
#!/bin/sh

# This sample JAMSHELL uses the SunOS on(1) command to execute a
# command string with an identical environment on another host.

# Set JAMSHELL = jamshell ! %
#
# where jamshell is the name of this shell file.
#
# This version handles up to -j6; after that they get executed
# locally.

case $1 in
1|4) on winken sh -c "$2";;
2|5) on blinken sh -c "$2";;
3|6) on nod sh -c "$2";;
*) eval "$2";;
esac
```

__TIMING_RULE__ and __ACTION_RULE__

The `__TIMING_RULE__` and `__ACTION_RULE__` can be set to the name of a rule for `bjam` to call **after** an action completes for a target. They both give diagnostic information about the action that completed. For `__TIMING_RULE__` the rule is called as:

```
rule timing-rule ( args * : target : start end user system )
```

And `__ACTION_RULE__` is called as:

```
rule action-rule ( args * : target : command status start end user system : output ? )
```

The arguments for both are:

- `args` Any values following the rule name in the `__TIMING_RULE__` or `__ACTION_RULE__` are passed along here.
- `target` The `bjam` target that was built.
- `command` The text of the executed command in the action body.

status	The integer result of the executed command.
start	The starting timestamp of the executed command as a ISO-8601 UTC value.
end	The completion timestamp of the executed command as a ISO-8601 UTC value.
user	The number of user CPU seconds the executed command spent as a floating point value.
system	The number of system CPU seconds the executed command spent as a floating point value.
output	The output of the command as a single string. The content of the output reflects the use of the <code>-px</code> option.



Note

If both variables are set for a target both are called, first `__TIMING_RULE__` then `__ACTION_RULE__`.

Modules

Boost Jam introduces support for modules, which provide some rudimentary namespace protection for rules and variables. A new keyword, "module" was also introduced. The features described in this section are primitives, meaning that they are meant to provide the operations needed to write Jam rules which provide a more elegant module interface.

Declaration

```
module expression { ... }
```

Code within the `{ ... }` executes within the module named by evaluating *expression*. Rule definitions can be found in the module's own namespace, and in the namespace of the global module as *module-name.rule-name*, so within a module, other rules in that module may always be invoked without qualification:

```
module my_module
{
    rule salute ( x ) { ECHO $(x), world ; }
    rule greet ( ) { salute hello ; }
    greet ;
}
my_module.salute goodbye ;
```

When an invoked rule is not found in the current module's namespace, it is looked up in the namespace of the global module, so qualified calls work across modules:

```
module your_module
{
    rule bedtime ( ) { my_module.salute goodnight ; }
}
```

Variable Scope

Each module has its own set of dynamically nested variable scopes. When execution passes from module A to module B, all the variable bindings from A become unavailable, and are replaced by the bindings that belong to B. This applies equally to local and global variables:


```

module A
{
    x = 1 ;
    rule f ( )
    {
        local y = 999 ; # becomes visible again when B.f calls A.g
        B.f ;
    }
    rule g ( )
    {
        ECHO $(y) ;      # prints "999"
    }
}
module B
{
    y = 2 ;
    rule f ( )
    {
        ECHO $(y) ; # always prints "2"
        A.g ;
    }
}

```

The only way to access another module's variables is by entering that module:

```

rule peek ( module-name ? : variables + )
{
    module $(module-name)
    {
        return $($(>)) ;
    }
}

```

Note that because existing variable bindings change whenever a new module scope is entered, argument bindings become unavailable. That explains the use of "\$(>)" in the peek rule above.

Local Rules

```
local rule rulename...
```

The rule is declared locally to the current module. It is not entered in the global module with qualification, and its name will not appear in the result of:

```
[ RULENAMES module-name ]
```

The RULENAMES Rule

```
rule RULENAMES ( module ? )
```

Returns a list of the names of all non-local rules in the given module. If *module* is omitted, the names of all non-local rules in the global module are returned.

The `VARNAMES` Rule

```
rule VARNAMES ( module ? )
```

Returns a list of the names of all variable bindings in the given module. If *module* is omitted, the names of all variable bindings in the global module are returned.



Note

This includes any local variables in rules from the call stack which have not returned at the time of the `VARNAMES` invocation.

The `IMPORT` Rule

`IMPORT` allows rule name aliasing across modules:

```
rule IMPORT ( source_module ? : source_rules *
             : target_module ? : target_rules * )
```

The `IMPORT` rule copies rules from the *source_module* into the *target_module* as local rules. If either *source_module* or *target_module* is not supplied, it refers to the global module. *source_rules* specifies which rules from the *source_module* to import; *target_rules* specifies the names to give those rules in *target_module*. If *source_rules* contains a name which doesn't correspond to a rule in *source_module*, or if it contains a different number of items than *target_rules*, an error is issued. For example,

```
# import m1.rule1 into m2 as local rule m1-rule1.
IMPORT m1 : rule1 : m2 : m1-rule1 ;
# import all non-local rules from m1 into m2
IMPORT m1 : [ RULENAMES m1 ] : m2 : [ RULENAMES m1 ] ;
```

The `EXPORT` Rule

`EXPORT` allows rule name aliasing across modules:

```
rule EXPORT ( module ? : rules * )
```

The `EXPORT` rule marks *rules* from the *source_module* as non-local (and thus exportable). If an element of *rules* does not name a rule in *module*, an error is issued. For example,

```
module X {
  local rule r { ECHO X.r ; }
}
IMPORT X : r : : r ; # error - r is local in X
EXPORT X : r ;
IMPORT X : r : : r ; # OK.
```

The `CALLER_MODULE` Rule

```
rule CALLER_MODULE ( levels ? )
```

`CALLER_MODULE` returns the name of the module scope enclosing the call to its caller (if *levels* is supplied, it is interpreted as an integer number of additional levels of call stack to traverse to locate the module). If the scope belongs to the global module, or if no such module exists, returns the empty list. For example, the following prints "{Y} {X}":

```

module X {
    rule get-caller { return [ CALLER_MODULE ] ; }
    rule get-caller's-caller { return [ CALLER_MODULE 1 ] ; }
    rule call-Y { return Y.call-X2 ; }
}
module Y {
    rule call-X { return X.get-caller ; }
    rule call-X2 { return X.get-caller's-caller ; }
}
callers = [ X.get-caller ] [ Y.call-X ] [ X.call-Y ] ;
ECHO {$(callers)} ;

```

The `DELETE_MODULE` Rule

```
rule DELETE_MODULE ( module ? )
```

`DELETE_MODULE` removes all of the variable bindings and otherwise-unreferenced rules from the given module (or the global module, if no module is supplied), and returns their memory to the system.



Note

Though it won't affect rules that are currently executing until they complete, `DELETE_MODULE` should be used with extreme care because it will wipe out any others and all variable (including locals in that module) immediately. Because of the way dynamic binding works, variables which are shadowed by locals will not be destroyed, so the results can be really unpredictable.

Miscellaneous

Diagnostics

In addition to generic error messages, `bjam` may emit one of the following:

```
warning: unknown rule X
```

A rule was invoked that has not been defined with an "actions" or "rule" statement.

```
using N temp target(s)
```

Targets marked as being temporary (but nonetheless present) have been found.

```
updating N target(s)
```

Targets are out-of-date and will be updated.

```
can't find N target(s)
```

Source files can't be found and there are no actions to create them.

```
can't make N target(s)
```

Due to sources not being found, other targets cannot be made.

```
warning: X depends on itself
```

A target depends on itself either directly or through its sources.

```
don't know how to make X
```

A target is not present and no actions have been defined to create it.

```
X skipped for lack of Y
```

A source failed to build, and thus a target cannot be built.

```
warning: using independent target X
```

A target that is not a dependency of any other target is being referenced with `$(<)` or `$(>)`.

```
X removed
```

BJam removed a partially built target after being interrupted.

Bugs, Limitations

For parallel building to be successful, the dependencies among files must be properly spelled out, as targets tend to get built in a quickest-first ordering. Also, beware of un-parallelizable commands that drop fixed-named files into the current directory, like `yacc(1)` does.

A poorly set `$(JAMSHELL)` is likely to result in silent failure.

Fundamentals

This section is derived from the official Jam documentation and from experience using it and reading the Jambase rules. We repeat the information here mostly because it is essential to understanding and using Jam, but is not consolidated in a single place. Some of it is missing from the official documentation altogether. We hope it will be useful to anyone wishing to become familiar with Jam and the Boost build system.

- Jam "rules" are actually simple procedural entities. Think of them as functions. Arguments are separated by colons.
- A Jam **target** is an abstract entity identified by an arbitrary string. The build-in `DEPENDS` rule creates a link in the dependency graph between the named targets.
- Note that the original Jam documentation for the built-in `INCLUDES` rule is incorrect: `INCLUDES targets1 : targets2` causes everything that depends on a member of *targets1* to depend on all members of *targets2*. It does this in an odd way, by tacking *targets2* onto a special tail section in the dependency list of everything in *targets1*. It seems to be OK to create circular dependencies this way; in fact, it appears to be the "right thing to do" when a single build action produces both *targets1* and *targets2*.
- When a rule is invoked, if there are `actions` declared with the same name as the rule, the actions are added to the updating actions for the target identified by the rule's first argument. It is actually possible to invoke an undeclared rule if corresponding actions are declared: the rule is treated as empty.
- Targets (other than `NOTFILE` targets) are associated with paths in the file system through a process called binding. Binding is a process of searching for a file with the same name as the target (sans grist), based on the settings of the target-specific `SEARCH` and `LOCATE` variables.
- In addition to local and global variables, jam allows you to set a variable on a target. Target-specific variable values can usually not be read, and take effect only in the following contexts:

- In updating actions, variable values are first looked up on the target named by the first argument (the target being updated). Because Jam builds its entire dependency tree before executing actions, Jam rules make target-specific variable settings as a way of supplying parameters to the corresponding actions.
- Binding is controlled *entirely* by the target-specific setting of the `SEARCH` and `LOCATE` variables, as described here.
- In the special rule used for header file scanning, variable values are first looked up on the target named by the rule's first argument (the source file being scanned).
- The "bound value" of a variable is the path associated with the target named by the variable. In build actions, the first two arguments are automatically replaced with their bound values. Target-specific variables can be selectively replaced by their bound values using the `bind` action modifier.
- Note that the term "binding" as used in the Jam documentation indicates a phase of processing that includes three sub-phases: *binding* (yes!), update determination, and header file scanning. The repetition of the term "binding" can lead to some confusion. In particular, the Modifying Binding section in the Jam documentation should probably be titled "Modifying Update Determination".
- "Grist" is just a string prefix of the form `<characters>`. It is used in Jam to create unique target names based on simpler names. For example, the file name `"test.exe"` may be used by targets in separate subprojects, or for the debug and release variants of the "same" abstract target. Each distinct target bound to a file called `"test.exe"` has its own unique grist prefix. The Boost build system also takes full advantage of Jam's ability to divide strings on grist boundaries, sometimes concatenating multiple gristed elements at the beginning of a string. Grist is used instead of identifying targets with absolute paths for two reasons:
 1. The location of targets cannot always be derived solely from what the user puts in a Jamfile, but sometimes depends also on the binding process. Some mechanism to distinctly identify targets with the same name is still needed.
 2. Grist allows us to use a uniform abstract identifier for each built target, regardless of target file location (as allowed by setting `ALL_LOCATE_TARGET`).
- When grist is extracted from a name with `$(var:G)`, the result includes the leading and trailing angle brackets. When grist is added to a name with `$(var:G=expr)`, existing grist is first stripped. Then, if `expr` is non-empty, leading `<s` and trailing `>s` are added if necessary to form an expression of the form `<expr2>`; `<expr2>` is then prepended.
- When Jam is invoked it imports all environment variable settings into corresponding Jam variables, followed by all command-line (`-s...`) variable settings. Variables whose name ends in `PATH`, `Path`, or `path` are split into string lists on OS-specific path-list separator boundaries (e.g. `":"` for UNIX and `","` for Windows). All other variables are split on space (`" "`) boundaries. Boost Jam modifies that behavior by allowing variables to be quoted.
- A variable whose value is an empty list or which consists entirely of empty strings has a negative logical value. Thus, for example, code like the following allows a sensible non-empty default which can easily be overridden by the user:

```
MESSAGE ?\= starting jam... ;
if $(MESSAGE) { ECHO The message is: $(MESSAGE) ; }
```

If the user wants a specific message, he invokes jam with `"-sMESSAGE=message text"`. If he wants no message, he invokes jam with `-sMESSAGE=` and nothing at all is printed.

- The parsing of command line options in Jam can be rather unintuitive, with regards to how other Unix programs accept options. There are two variants accepted as valid for an option:
 1. `-xvalue`, and
 2. `-x value`.

History

- 3.1.18 After years of bjam developments.. This is going to be the last unbundled release of the 3.1.x series. From this point forward bjam will only be bundled as part of the larger Boost Build system. And hence will likely change name at some point. As a side effect of this move people will get more frequent release of bjam (or whatever it ends up being called).

- New built-ins, MD5, SPLIT_BY_CHARACTERS, PRECIOUS, PAD, FILE_OPEN, and UPDATE_NOW. -- *Vladimir P.*
- Ensure all file descriptors are closed when executing actions complete on *nix. -- *Noel B.*
- Fix warnings, patch from Mateusz Loskot. -- *Vladimir P.*
- Add KEEP_GOING var to programatically override the '-q' option. -- *Vladimir P.*
- Add more parameters, up to 19 from 9, to rule invocations. Patch from Jonathan Biggar. -- *Vladimir P.*
- Print failed command output even if the normally quite '-d0' option. -- *Vladimir P.*
- Build of bjam with vc10, aka Visual Studio 2010. -- *Vladimir P.*
- More macros for detection of OSPLAT, patch from John W. Bito. -- *Vladimir P.*
- Add PARALLELISM var to programatically override the '-j' option. -- *Vladimir P.*
- Tweak doc building to allow for PDF generation of docs. -- *John M.*

3.1.17 A year in the making this release has many stability improvements and various performance improvements. And because of the efforts of Jurko the code is considerably more readable!

- Reflect the results of calling bjam from Python. -- *Rene R.*
- For building on Windows: Rework how arguments are parsed and tested to fix handling of quoted arguments, options arguments, and arguments with "=". -- *Rene R.*
- Try to work around at least one compiler bug with GCC and variable aliasing that causes crashes with hashing file cache entries. -- *Rene R.*
- Add -Wc,-fno-strict-aliasing for QCC/QNX to avoid the same aliasing crashes as in the general GCC 4.x series (thanks to Niklas Angare for the fix). -- *Rene R.*
- On Windows let the child bjam commands inherit stdin, as some commands assume it's available. -- *Rene R.*
- On Windows don't limit bjam output to ASCII as some tools output characters in extended character sets. -- *Rene R.*
- Isolate running of bjam tests to individual bjam instances to prevent possible spillover errors from one test affecting another test. Separate the bjam used to run the tests vs. the bjam being tested. And add automatic re-building of the bjam being tested. -- *Rene R.*
- Fix some possible overrun issues revealed by Fortify build. Thanks to Steven Robbins for pointing out the issues. -- *Rene R.*
- Handle \n and \r escape sequences. -- *Vladimir P.*
- Minor edits to remove -Wall warnings. -- *Rene R.*
- Dynamically adjust pwd buffer query size to allow for when PATH_MAX is default defined instead of being provided by the system C library. -- *Rene R.*
- Minor perf improvement for bjam by replacing hash function with faster version. Only 1% diff for Boost tree. -- *Rene R.*
- Updated Boost Jam's error location reporting when parsing Jamfiles. Now it reports the correct error location information when encountering an unexpected EOF. It now also reports where an invalid lexical token being read started instead of finished which makes it much easier to find errors like unclosed quotes or curly braces. -- *Jurko G.*

- Removed the `-xarch=generic` architecture from `build.jam` as this option is unknown so the Sun compilers on Linux. - *Noel B.*
- Fixed a bug with `T_FATE_ISTMP` getting reported as `T_FATE_ISTMP` & `T_FATE_NEEDTMP` at the same time due to a missing break in a switch statement. -- *Jurko G.*
- Fixed a Boost Jam bug causing it to sometimes trigger actions depending on targets that have not been built yet. -- *Jurko G.*
- Added missing documentation for Boost Jam's `:T` variable expansion modifier which converts all back-slashes (`\`) to forward slashed (`/`). -- *Jurko G.*
- Added Boost Jam support for executing command lines longer than 2047 characters (up to 8191) characters when running on Windows XP or later OS version. -- *Jurko G.*
- Fixed a Boost Jam bug on Windows causing its SHELL command not to work correctly with some commands containing quotes. -- *Jurko G.*
- Corrected a potential memory leak in Boost Jam's `builtin_shell()` function that would appear should Boost Jam ever start to release its allocated string objects. -- *Jurko G.*
- Made all Boost Jam's ECHO commands automatically flush the standard output to make that output more promptly displayed to the user. -- *Jurko G.*
- Made Boost Jam tests quote their `bjam` executable name when calling it allowing those executables to contain spaces in their name and/or path. -- *Jurko G.*
- Change `execunix.c` to always use `fork()` instead of `vfork()` on the Mac. This works around known issues with `bjam` on PPC under Tiger and a problem reported by Rene with `bjam` on x86 under Leopard. -- *Noel B.*
- Corrected a bug in Boost Jam's base `Jambase` script causing it to trim the error message displayed when its `boost-build` rule gets called multiple times. -- *Jurko G.*
- When importing from Python into a module with empty string as name, import into root module. -- *Vladimir P.*
- Patch for the `NORMALIZE_PATH` builtin Boost Jam rule as well as an appropriate update for the `path.jam` Boost Build module where that rule was being used to implement path join and related operations. -- *Jurko G.*
- Fixed a bug causing Boost Jam not to handle target file names specified as both short and long file names correctly. - *Jurko G.*
- Relaxed test, ignoring case of drive letter. -- *Roland S.*
- Implemented a patch contributed by Igor Nazarenko reimplementing the `list_sort()` function to use a C `qsort()` function instead of a hand-crafted merge-sort algorithm. Makes some list sortings (e.g. `1,2,1,2,1,2,1,2, ...`) extremely faster, in turn significantly speeding up some project builds. -- *Jurko G.*
- Fixed a bug with `bjam` not handling the `"` root Windows path correctly without its drive letter being specified. -- *Jurko G.*
- Solved the problem with child process returning the value 259 (Windows constant `STILL_ACTIVE`) causing `bjam` never to detect that it exited and therefore keep running in an endless loop. -- *Jurko G.*
- Solved the problem with `bjam` going into an active wait state, hogging up processor resources, when waiting for one of its child processes to terminate while not all of its available child process slots are being used. -- *Jurko G.*
- Solved a race condition between `bjam`'s output reading/child process termination detection and the child process's output generation/termination which could have caused `bjam` not to collect the terminated process's final output. -- *Jurko G.*
- Change from `vfork` to `fork` for executing actions on Darwin to improve stability. -- *Noel B.*

- Code reformatting and cleanups. -- *Jurko G.*
- Implement ISFILE built-in. -- *Vladimir P.*

3.1.16 This is mostly a bug fix release.

- Work around some Windows CMD.EXE programs that will fail executing a totally empty batch file. -- *Rene R.*
- Add support for detection and building with `vc9`. -- *John P.*
- Plug memory leak when closing out actions. Thanks to Martin Kortmann for finding this. -- *Rene R.*
- Various improvements to `__TIMING_RULE__` and `__ACTION_RULE__` target variable hooks. -- *Rene R.*
- Change `JAMDATE` to use common ISO date format. -- *Rene R.*
- Add test for result status values of simple actions, i.e. empty actions. -- *Rene R.*
- Fix buffer overrun bug in expanding `@()` subexpressions. -- *Rene R.*
- Check empty string invariants, instead of assuming all strings are allocated. And reset strings when they are freed. -- *Rene R.*
- Add `OSPLAT=PARISC` for HP-UX PA-RISC. -- *Boris G.*
- Make quietly actions really quiet by not printing the command output. The output for the quietly actions is still available through `__ACTION_RULE__`. -- *Rene R.*
- Switch intel-win32 to use static multi thread runtime since the single thread static runtime is no longer available. -- *Rene R.*
- When setting `OSPLAT`, check `__ia64` macro. -- *Boris G.*
- Get the unix timing working correctly. -- *Noel B.*
- Add `-fno-strict-aliasing` to compilation with gcc. Which works around GCC-4.2 crash problems. -- *Boris G.*
- Increased support for Python integration. -- *Vladimir P., Daniel W.*
- Allow specifying options with quotes, i.e. `--with-python=xyz`, to work around the CMD shell using `=` as an argument separator. -- *Rene R.*
- Add values of variables specified with `-s` to `.EVNRION` module, so that we can override environment on command line. -- *Vladimir P.*
- Make `NORMALIZE_PATH` convert `\` to `/`. -- *Vladimir P.*

3.1.15 This release sees a variety of fixes for long standing Perforce/Jam problems. Most of them relating to running actions in parallel with the `-jN` option. The end result of the changes is that running parallel actions is now reliably possible in Unix and Windows environments. Many thanks to Noel for joining the effort, to implement and fix the Unix side of stuff.

- Add support for building bjam with pgi and pathscale toolsets. -- *Noel B.*
- Implement running action commands through pipes (`-p` option) to fix jumbled output when using parallel execution with `-j` option. This is implemented for Unix variants, and Windows (Win32/NT). -- *Rene R., Noel B.*
- Add "sun" as alias to Sun Workshop compiler tools. -- *Rene R.*

- Set MAXLINE in jam.h to 23k bytes for AIX. The piecemeal archive action was broken with the default MAXLINE of 102400. Because the AIX shell uses some of the 24k default buffer size for its own use, I reduced it to 23k. -- *Noel B.*
- Make use of output dir options of msvc to not pollute src dir with compiled files. -- *Rene R.*
- A small fix, so -d+2 will always show the "real" commands being executed instead of casually the name of a temporary batch file. -- *Roland S.*
- Add test to check 'bjam -n'. -- *Rene R.*
- Add test to check 'bjam -d2'. -- *Rene R.*
- Bring back missing output of -n option. The -o option continues to be broken as it has been for a long time now because of the @ file feature. -- *Rene R.*
- Update GC support to work with Boehm GC 7.0. -- *Rene R.*
- Revert the BOOST_BUILD_PATH change, since the directory passed to boost-build should be first in searched paths, else project local build system will not be picked correctly. The order had been changed to allow searching of alternate user-config.jam files from boost build. This better should be done with --user-config= switch or similar. -- *Roland S.*
- Initial support for defining action body from Python. -- *Vladimir P.*
- Implement @() expansion during parse phase. -- *Rene R.*
- Define OSPLAT var unconditionally, and more generically, when possible. -- *Rene R.*
- Fix undeclared INT_MAX on some platforms, i.e. Linux. -- *Rene R.*
- Modified execunix.c to add support for terminating processes that consume too much cpu or that hang and fail to consume cpu at all. This in support of the bjam -lx option. -- *Noel B.*
- Add internal dependencies for multi-file generating actions to indicate that the targets all only appear when the first target appears. This fixes the long standing problem Perforce/Jam has with multi-file actions and parallel execution (-jN). -- *Rene R.*
- Add test of -l limit option now that it's implemented on windows and unix. -- *Rene R.*
- Add test for no-op @() expansion. -- *Rene R.*
- Handle invalid formats of @() as doing a straight substitution instead of erroring out. -- *Rene R.*
- Various fixes to compile on SGI/Irix. -- *Noel B.*
- Add output for when actions timeout with -lN option. -- *Rene R., Noel B.*
- Add needed include (according to XOPEN) for definition of WIFEXITED and WEXITSTATUS. -- *Markus S.*