NAME

jam – **make**(1) redux

SYNOPSIS

jam [**-a**] [**-n**] [**-v**] [**-d** *debug*] [**-f** *Jambase ...*] [**-j** *jobs*] [**-t** *target ...*] [*target ...*]

DESCRIPTION

Jam recursively builds target files from their source files, using two files to define the dependency graph and the updating actions for all targets. The file **Jambase** (usually located in /usr/local/lib) defines rules, and the file **Jamfile** (located in the current directory) lists the targets and sources in terms of those rules. **Jam** does not need to rely on suffix-driven implicit rules or directory contents. A **Jambase** is provided with **jam**; the user supplies the **Jamfile**.

See **Jamfile**(5) for information on writing Jamfiles. This manual page describes the program that interprets **Jambase** and **Jamfile**.

OPTIONS

If *target* is provided on the command line, **jam** attempts to build that target; otherwise **jam** attempts to build the target *all*.

Jam supports the following options:

-a Build all targets, even if they are up-to-date.

-d<*n*> Set the debug level to <*n*>. Interesting values are:

0 Emit only errors
1 Emit update action tracing (default)
2 Emit update commands
3 Produce dependency information
4 Show modification times of bound files
5 Show rule invocation
6-9 debugging

-f<*file*>
Read <*file*> instead of **Jambase**.

-j<*jobs*>
Run up to <*jobs*> shell commands concurrently (UNIX only). The default is 1.

-n Don't actually execute the updating actions, but do everything else. This implies **-d***2*.

-t<*target*>
Rebuild <*target*>, even if it is up-to-date.

-v Print the version of **jam** and exit.

THE JAM LANGUAGE

The **jam** language supports defining rules, invoking rules, and setting variables. It also has a few flow-of-control statements. **Jambase** and **Jamfile** share this language.

Lexical Features

Jam 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.

Jam requires whitespace (blanks, tabs, or newlines) to surround all tokens, including the colon (:) and semicolon (;) tokens. This is because **jam** runs on many platforms and no characters, save whitespace, are uncommon in the file names on all of those platforms.

Targets

Targets are files to be updated and sources are the files used in updating those targets. Collectively, they are just referred to as "targets". A target is simply a file name, either rooted or relative to the directory of **jam**'s invocation. The special syntax, *file(member)*, refers to an **ar**(1) library member. The special syntax, *<grist>file*, perturbs a file name to distinguish it from other files with the same name. The *<grist>* is stripped from the name during binding (q.v., below).

Rules

Jam's basic entity is called a rule, which is used to relate targets to their sources. A rule is defined in two parts: the **jam** statements to execute when the rule is invoked (essentially a procedure call), and the actions (shell commands) to execute in order to update the targets of the rule. A rule may have a procedure definition, actions, or both.

The **jam** statements for defining and invoking rules are as follows. *<targets>* and *<sources>* are lists of file names; *<statements>* are **jam** statements; and *<string>* is a shell script:

rule *<rulename>* { *<statements>* }

actions [*modifiers*] *<rulename>* { *<string>* }

<rulename> *<targets>* [: *<sources>*] ;

The first form defines a rule's procedure; the second defines the rule's updating actions; the third invokes the rule. Redefining a rule's procedure or actions replaces the previous definition.

Invoking a rule executes the procedure for the rule (if any) and associates any update actions for the targets. More than one update action may be associated with a target: the actions are executed in the order in which they are added.

In both the rule's procedure definition and the rule's actions, the special variables $(<) and $(>) refer to the *<targets>* and *<sources>* given at rule invocation. However, in the rule's actions, $(<) and $(>) refer to the *<targets>* and *<sources>* after they have been bound by the binding phase (q.v., below). **Jam** issues a warning if $(<) or $(>) have elements not in the dependency graph.

The following action *modifiers* are understood:

actions existing
$(>) includes only targets currently existing.

actions ignore
The return status of the shell commands is ignored.

actions piecemeal
The shell commands are repeatedly invoked with a subset of $(>) small enough to fit in a command buffer.

actions quietly
The action is not echoed to the standard output.

actions together
The $(>) from multiple instances of the same action on the same target are glommed together.

actions updated
$(>) includes only targets marked for updating.

Built-in Rules

Jam has ten built-in rules, none of which have updating actions:

ALWAYS *<targets>* ;
Rebuilds *<targets>*, even if they are up-to-date.

DEPENDS *<targets>* : *<sources>* ;
Makes *<sources>* dependencies of *<targets>*.

ECHO *<args>* ;
Blurts out the message *<args>* to stdout.

EXIT *<args>* ;
Blurts out the message *<args>* to stdout and then exits with a failure status.

INCLUDES *<targets>* : *<sources>* ;
Makes *<sources>* dependencies of anything of which *<targets>* are dependencies.

LEAVES *<targets>* ;
Makes each of *<targets>* depend only on its leaf sources, and not on any intermediate targets. Its leaf sources are those dependencies without any dependencies themselves.

NOCARE *<targets>* ;
Marks *<targets>* as possibly being bogus.

NOTFILE *<targets>* ;
Marks *<targets>* as not being files.

NOUPDATE *<targets>* ;
Causes the timestamps of *<targets>* to be ignored: either the target exists or it doesn't. If it exists, it is considered eternally old.

TEMPORARY *<targets>* ;
Marks *<targets>* as temporary.

The *ALWAYS*, *LEAVES*, *NOCARE*, *NOTFILE*, *NOUPDATE*, and *TEMPORARY* affect only the binding phase (q.v.).

Flow-of-Control

Jam has several simple flow-of-control statements:

include *<a>* ;

for *<a>* in *<args>* { *<statements>* }

switch *<a>* { case *<v1>* : *<statements>* ; case *<v2>* : *<statements>* ; ... }

if *<cond>* { *<statements>* } [else { *<statements>* }]

The **include** statement includes the named file. The file is bound like regular targets (see **Binding**, below), but unlike regular targets the include file cannot be built.

The **for** loop executes *<statements>* for each value in *<args>*, setting the variable *<a>* to the value.

The **switch** statement executes zero or one of the enclosed *<statements>*, depending on which value *<a>* matches. The *<v>* values are not variable-expanded. The *<v>* values may include the following wildcards:

?	match any single character
*	match zero or more characters
[*<chars>*]	match any single character in *<chars>*

The **if** statement does the obvious; the **else** clause is optional. *<cond>* is built of:

<a>	true if *<a>* is a non-zero-length string
<a> = *<b>*	strings equal
<a> != *<b>*	strings not equal
<a> < *<b>*	string less than
<a> <= *<b>*	string less than or equal to
<a> > *<b>*	string greater than
<a> >= *<b>*	string greater than or equal to

! *<cond>*	condition not true
<cond> && *<cond>*	conjunction
<cond> || *<cond>*	disjunction

(*<cond>*) grouping

In comparisons, the arguments may (through variable expansion) be more than one token, but only the first token takes part in the comparison. If, through variable expansion, the argument is zero tokens, a single token of a zero-length string is used instead.

Variables

Jam variables are lists of strings, with zero or more elements. An undefined variable is indistinguishable from a variable whose value is an empty list. Variables are either global or target-specific. All variables are referenced as $(VARIABLE).

A variable is defined with:

<variable> = *<values>* ;

<variable> += *<values>* ;

<variable> on *<targets>* = *<values>* ;

<variable> on *<targets>* += *<values>* ;

<variable> default = *<values>* ;

The first two forms set *<variable>* globally. The third and forth forms set a target-specific variable, where *<variable>* takes on a value only during the binding and updating *<targets>*. The **=** operator replaces any previous value of *<variable>* with *<values>*; the **+=** operation appends *<values>* to any previous value. The final form sets *<variable>* globally, but only if it was previously unset.

On program start-up, **jam** imports the environment variable settings into **jam** variables. Environment variables are split at blanks with each word becomming an element in the variable's list value. Environment variables whose names end in **PATH** are split at colons ("**:**"). **Jam** variables are not re-exported to the shell that executes the updating actions, but the updating actions can reference **jam** variables with $(VARIABLE).

Variable Expansion

Before executing a statement, **jam** 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.

Modifiers to a variable are of two varieties: sub-element selection and file name editing. They are:

[*<n>*] Select only element number *<n>* (starting at 1). If the variable contains fewer than *<n>* elements, the result is a zero-element list.

[*<n>*-*<m>*]
Select only elements number *<n>* through *<m>*.

[*<n>*-] Select only elements number *<n>* through the last.

:G=*<grist>*
Replace the grist of the file name with *<grist>*.

:D=*<path>*
Replace directory component of file name with *<path>*.

:B=*<base>*
Replace the base part of file name with *<base>*.

:S=*<suf>*
Replace the suffix of file name with *<suf>*.

:M=*<mem>*
Replace the archive member name with *<mem>*.

:R=*<root>*
Prepend *<root>* to the whole file name, if not already rooted.

:*<components>*
Remove components not listed; components is one or more of **GDBSM**.

OPERATION

Jam has three phases of operation: parsing, binding, and updating.

Parsing

Jam parses the **Jambase** file, which includes **Jamfile**. The results of parsing are: the dependency graph of targets; update actions associated with the targets; and variables set to specific values.

Binding

After parsing, **jam** recursively descends the dependency graph, attempting to locate each target file and determine if it is in need of updating. If **jam** detects a cycle in the graph, it issues a warning.

By default, a target is located at the actual path of the target, relative to the directory of **jam**'s invocation. If the special variable $(LOCATE) is set to a directory name, **jam** prepends that directory name to the target; else if the special variable $(SEARCH) is set to a directory list, **jam** searches along the directory list for the target file, and if the file is found prepends the directory name to the target. If the target name has a rooted directory component then $(SEARCH) and $(LOCATE) do not apply: the target is located at the actual path of the target. If a target is marked as not being a file (using the built-in rule NOTFILE), it is left unbound to a file name.

After binding each target, **jam** determines whether the target needs updating, and marks the target if necessary for the updating phase. A target is marked for updating for any of these three reasons:

It is missing.

Its filesystem modification time is older than any of its sources.

Any of its sources are marked for updating.

This basic behavior can be modified applying (usually one of) the following six built-in rules to the target:

ALWAYS
The target is always updated.

LEAVES
The target is only updated if it is missing or if its leaf sources are newer. Leaf sources are those dependencies of the target that have no dependencies themselves.

NOCARE
: The target is ignored if it is missing and has no updating actions. Normally, **jam** issues a warning and skips other targets that depend on missing targets without updating actions.

TEMPORARY
: If the target is missing, then its parent's modification time is used when comparing against sources.

NOTFILE
: The target is only updated if any of its sources are marked for updating.

NOUPDATE
: The target is only updated if it is missing. Also, if it exists, it will appear eternally old; that is, older than anything that depends on it.

If a target is a source file that includes other files, **jam** scans the source file for header file include lines. It scans the file by matching each line against a **regexp**(3) pattern that has ()'s surrounding the included file name. The pattern is provided by the user through the special variable $(HDRSCAN) (see **HDRPATTERN** in **Jambase** for an example). 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. The rule invoked is named by the special variable $(HDRRULE). **Jam** only scans files if $(HDRSCAN) is set, and $(HDRSCAN) is normally set target-specific.

Between binding and updating, **jam** announces the number of targets to be updated.

Updating

After binding, **jam** 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 targets which depend on it are skipped.

(UNIX only). The **-j** flag instructs **jam** to build more than one target at a time. If there are multiple actions on a single target, they are run sequentially.

(UNIX only). The special variable $(JAMSHELL) gives **jam** a command execution shell to be used instead of /bin/sh. This variable's value must be a multi-element list, corresponding to the argument vector for the command shell. An element "**%**" is replaced with the command string to execute. An element "**!**" is replaced with the multiprocess slot number, which is (inclusively) between 1 and the maximum number of concurrent jobs specified with the **-j** flag (default 1). If no element of the list is "**%**", the command string is tacked on as the last argument. The default value is: "/bin/sh -c %".

DIAGNOSTICS

In addition to generic error messages, **jam** 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 does is not a dependency of any other target is being referenced with $(<) or $(>).

X removed

Jam removed a partially built target after being interrupted.

FILES

/usr/local/lib/Jambase
Jamfile

BUGS, LIMITATIONS

Because the **include** statement works by pushing a new file in the input stream of the scanner rather than recursively invoking the parser on the new file, multiple include statements in a rule's procedure causes the files to be included in reverse order.

If the **include** statement appears inside an **if** block, the parser's attempt to find the **else** will cause the text of the included file to appear after the first token following the statement block. This is rarely what is intended.

In a rule's actions, only $(<) and $(>) refer to the bound file names: all other variable references get the unbound names.

With the **-j** flag, errors from failed commands can get staggeringly mixed up. Also, because targets tend to get built in a quickest-first ordering, dependency information must be quite exact. Finally, beware of parallelizing 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.

SEE ALSO

Jambase(5), **Jamfile**(5)