NAME

Jamfile – per-directory **jam**(1) instructions

DESCRIPTION

This manual page gives instructions and examples for writing a **Jamfile**, a file that tells the build tool **jam** what to build. It consists of invocations of rules defined by the **jam** boilerplate, **Jambase**. **Jambase** itself defines rules that provide roughly **make**(1)-like functionality.

The first section is a one-page overview of **jam**, to introduce the syntax. The remaining sections are examples and discussion.

This description goes with **Jam** Release 2.0.

JAM OVERVIEW

Invocation

The examples in the following sections are parts of a **Jamfile**. Once you have written a **Jamfile**, you can invoke **jam** to build things. The simplest syntax is:

```
jam [-n]
```

The **-n** tells **jam** to do a verbose dry-run.

Rules

The **jam** language consists mostly of rule invocations. A rule is invoked with *targets* and *sources*. That is,

```
Rule targets : sources ;
```

Each rule definition has potentially two parts: the rule procedure and the rule's actions. The procedure is just more **jam** statements to interpret when the rule is invoked. The actions are the shell commands to execute when the targets are to be updated.

The **Jambase** consists mostly of rule definitions, and your **Jamfile** will contain mostly rule invocations.

Variables

Jam has two types of variables: global ones and target-specific ones. The latter take effect only when the named target is being bound, scanned for header files, or updated. Target-specific variables enable generic actions (like the *Cc* rule's call to the C compiler) to be used for different targets. Only the target-specific variables (HDRS, CCFLAGS, etc.) vary.

The **Jambase** rules use variables in three ways:

Procedure Input
: If a variable is procedure input, it must be set before invoking the rule. This is sometimes used to pass additional parameters to the rule (beyond the targets and sources), and is sometimes used for conditional definitions (e.g., if the RANLIB variable is set, invoke the *Ranlib* rule). When we describe the variables in each section, we mark procedure input variables with a *.

Procedure Output
: A rule procedure may set variables for later use. Except where noted, all rules set these variables specific to the rule's targets. Sometimes the variable are later used during the binding and header file scanning phase, and sometimes they are used by the actions.

Actions Input
: The shell commands to update a target often contain variable references. These variables may be set globally or target-specific. The latter take precedence.

Often, a variable will be used in more than just way: a variable set by a rule procedure may quite likely be used by the actions. In some cases, a variable that is input to a rule procedure may be output (target-specific) as well. This has the effect of freezing the variable's value for the target.

Whitespace Note

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.

BUILDING EXECUTABLES AND LIBRARIES

The following rules compile source files and build executables and libraries.

Main Rule

The *Main* rule compiles source files and links the resulting objects into an executable. For example:

```
Main myprog : main.c util.c ;
```

This compiles main.c and util.c and links main.o and util.o into myprog. As with all rules that compile source files, *Main* handles header file dependencies automatically.

Library Rule

The *Library* rule compiles source files, archives the resulting object files into a library, and then deletes the object files. For example:

```
Library libstring.a : strcmp.c strcpy.c strlen.c ;
Library libtree.a : treemake.c treetrav.c ;
```

This compiles five source files, archives three of the object files into libstring.a and the other two into libtree.a. Once the objects are safely in the libraries, the objects are deleted.

LinkLibraries Rule

To tell **jam** to link executables against libraries, you use the *LinkLibraries* rule. For example:

```
Main myprog : main.c util.c ;
LinkLibraries myprog : libstring.a libtree.a ;

Library libstring.a : strcmp.c strcpy.c strlen.c ;
Library libtree.a : treemake.c treetrav.c ;
```

The *LinkLibraries* rule does two things: it makes the libraries dependencies of the executable, so that they get built first; and it makes the libraries show up on the command line that links the executable. The ordering of the lines above is not important, because **jam** builds targets in the order that they are needed.

You can put multiple libraries on a single invocation of the *LinkLibraries* rule, or you can provide them in multiple invocations. In both cases, the libraries appear on the link command line in the order in which they were encountered. You can also provide multiple executables to the *LinkLibraries* rule, if they need the same libraries.

Variables

The following variables control the linking of executables and the archiving of libraries.

```
$(AR)                archive command (ar ru)
$(EXEMODE)         * default value for MODE (711)
$(LINK)              linker command (cc)
$(LINKFLAGS)         linker flags ()
$(LINKLIBS)          additional libraries that aren't dependencies ()
$(MODE)              permission on target
$(RANLIB)            name of ranlib program, if any (ranlib)
```

Main sets a target-specific MODE to the current value of $(EXEMODE).

The difference between and the arguments to *LinkLibraries* and the value of $(LINKLIBS) is that the former are expected to be real, buildable libraries, while the latter are just handed without inspection to the $(LINK) command. The ordering on the link command line is first *LinkLibraries* and then $(LINKLIBS). For example:

```
Main xprog : xprog.c ;
LinkLibraries xprog : libxutil.a ;
LINKFLAGS on xprog = -Bstatic ;
LINKLIBS on xprog = -lXext -lX11 ;

Library libxutil.a : xtop.c xbottom.c xutil.c ;
```

This example uses the **jam** syntax "variable *on* target" to set a target-specific variable. In this way, only xprog will be linked with these special $(LINKFLAGS) and $(LINKLIBS), even if other executables were going to be built by the same **Jamfile**. The actual link command line would look like this:

```
cc -Bstatic -o xprog xprog.o libuxtil.a -lXext -lX11
```

Note that the default link command is cc.

COMPILING

Compiling of source files occurs normally as a byproduct of the *Main* or *Library* rules. If you want to control the compiling process with finer granularity, you can use the rules described here. *Main* and *Library* use these rules.

Objects Rule

The *Main* or *Library* rules call the *Objects* rule on each of their source files. You can also call *Objects* directly. For example:

```
Objects a.c b.c c.c ;
```

This compiles a.c into a.o, b.c into b.o, etc.

Object Rule

Objects gets its work done by calling the *Object* rule on each of the source files, making the assumption that the object name itself will be the source file name, with the suffix replaced appropriately. To compile a single source file directly, use the *Object* rule. For example:

```
Object foo.o : foo.c ;
```

The *Object* rule doesn't require that the object name bear any relationship to the source. It is thus possible to compile the same file into different objects. For example:

```
Object a.o : foo.c ;
Object b.o : foo.c ;
Object c.o : foo.c ;
```

This compiles foo.c (three times) into a.o, b.o, and c.o. Later examples show how this is useful.

The *Object* rule looks at the suffix of the source file and calls the appropriate rules to do the actual compiling. This invariably involves a call to the *Cc* to turn the .c into a .o, but may also require a call to other rules to turn the source file into a .c. Thus the *Object* rule is responsible for the generation of an object file from any type of source. For example:

```
Object grammar.o : grammar.y ;
Object scanner.o : scanner.l ;
Object fastf.o : fastf.f ;
Object util.o : util.c ;
```

In addition to calling the compiling rule, *Object* sets up a bunch of variables specific to the source and target files. These are discussed below.

Cc, C++, Yacc, Lex, Fortran, As, etc. Rules

The *Object* calls compiling rules specific to the suffix of the source file. Because the extra work done by the *Object* rule, it is not always useful to call the compiling rules directly. But the adventurous user might attempt it. For example:

```
Yacc grammar.c : grammar.y ;
Lex scan.c : scan.l ;
Cc prog.o : prog.c ;
```

These examples individually run **yacc**(1), **lex**(1), and the C compiler on their sources.

UserObject Rule

Any files with suffixes not understood by the *Object* rule are passed to the *UserObject* rule. The default definition of *UserObject* simply emits a warning that the suffix is not understood. This rule definition is intended to be replaced with one that recognizes the suffix. For example:

```
rule UserObject
{
    switch $(>)
    {
    case *.s    : As $(<) : $(>) ;
    case *      : ECHO "unknown suffix on" $(>) ;
    }
}

rule As
{
    DEPENDS $(<) : $(>) ;
}

actions As
{
    as -o $(<) $(>)
}

Library libsys.a : alloca.s memcpy.s ;
```

It should be mentioned that this example is contrived, in that the **.s** suffix is already handled by *Object*.

LibraryFromObjects Rule

Sometimes the *Library* rule's straightforward compiling of source into object modules to be archived isn't flexible enough. The *LibraryFromObjects* rule does the archiving (and deleting) job of the *Library* rule, but not the compiling. The user can make use of the *Objects* or *Object* rule for that. For example:

```
LibraryFromObjects libfoo.a : max.o min.o ;
Object max.o : maxmin.c ;
Object min.o : maxmin.c ;
ObjectCcFlags max.o : -DUSEMAX ;
ObjectCcFlags min.o : -DUSEMIN ;
```

This compiles the same source file into two different objects, with different compile flags, and archives them. The *ObjectCcFlags* rule is described shortly.

MainFromObjects Rule

Similar to *LibraryFromObjects*, *MainFromObjects* does the linking part of the *Main* rule, but not the compiling. For example:

```
MainFromObjects w : w.o ;
MainFromObjects uptime : uptime.o ;
Object w.o : uptime.c ;
Object uptime.o : uptime.c ;
ObjectCcFlags w.c : -DW_CODE ;
```

This compiles two different programs, w and uptime, from the same source file with different C compiler flags. Another use of *MainFromObjects* is when there are no objects at all, and everything is to be loaded from libraries. For example:

```
MainFromObjects testprog ;
LinkLibraries testprog : libprog.a ;
Library libprog.a : main.c util.c ;
```

This generates a link command that looks like this:

```
cc -o testprog libprog.a
```

Linking purely from libraries is something that doesn't work everywhere: it depends on the symbol "main" being undefined when the linker encounters the library that contains the definition of "main".

Variables

The following variables control the compiling of source files.

```
$(C++)                The C++ Compiler (gcc)
$(C++FLAGS)         * C++ compiler flags()
$(CC)                 The C Compiler (cc)
$(CCFLAGS)          * C compiler flags()
$(HDRS)             * non-standard header directories ()
$(LEX)                The Lex program (lex)
$(OPTIM)              optimization flag, if desired (-O)
$(STDHDRS)          * standard header directories (/usr/include)
$(SUBDIRC++FLAGS)   * Per-directory C++FLAGS
$(SUBDIRCCFLAGS)    * Per-directory CCFLAGS
$(SUBDIRHDRS)       * Per-directory HDRS
$(YACC)               The Yacc program (yacc -d)
```

The *Cc* rule sets a target-specific $(CCFLAGS) to the current value of $(CCFLAGS) and $(SUBDIRCCFLAGS). Similarly for the *C++* rule. The *Object* rule sets a target-specific $(HDRS) to the current value of $(HDRS) and $(SUBDDIRHDRS).

$(CC), $(C++), $(CCFLAGS), $(C++FLAGS), $(OPTIM), and $(HDRS) all affect the compiling of C and C++ files. $(OPTIM) is separate from $(CCFLAGS) and $(C++FLAGS) so they can be set independently.

$(HDRS) lists the directories to search for header files, and it is used in two ways: first, it is passed to the C compiler (with the flag **-I** prepended); second, it is used by *HdrRule* to locate the header files whose names were found when scanning source files. $(STDHDRS) lists the header directories that the C compiler already knows about. It does not need passing to the C compiler, but is used by *HdrRule*.

Note that these variables, if set as target-specific variables, must be set on the target, not the source file. The target file in this case is the object file to be generated. For example:

```
Library libximage.a : xtiff.c xjpeg.c xgif.c ;

HDRS on xjpeg.o = /usr/local/src/jpeg ;
CCFLAGS on xtiff.o = -DHAVE_TIFF ;
```

This can be done more easily with the rules that follow.

ObjectCcFlags, ObjectC++Flags, ObjectHdrs Rules

$(CCFLAGS), $(C++FLAGS) and $(HDRS) can be manipulated directly, but there are rules that allow these variables to be set by referring to the original source file name, rather than to the derived object file name. *ObjectCcFlags* adds object-specific flags to the $(CCFLAGS) variable, *ObjectC++Flags* adds object-specific flags to the $(C++FLAGS) variable, and *ObjectHdrs* add object-specific directories to the $(HDRS) variable. For example:

```
Main xviewer : viewer.c ;
ObjectCcFlags viewer.c : -DXVERSION ;
ObjectHdrs viewer.c : /usr/include/X11 ;
```

Actually, the file suffix (**.c** in this case) is ignored: the rules know to refer to the object.

SubDirCcFlags, SubDirC++Flags, SubDirHdrs Rules

These rules set the values of $(SUBDIRCCFLAGS), $(SUBDIRC++FLAGS) and $(SUBDIRHDRS), which are used by the *Cc*, *C++*, and *Object* rules when setting the target-specific values for $(CCFLAGS), $(C++FLAGS) and $(HDRS). The *SubDir* rule clears these variables out, and thus they provide directory-specific values of $(CCFLAGS), $(C++FLAGS) and $(HDRS). For example:

```
SubDir TOP src util ;

SubDirHdrs $(TOP)/src/hdr ;
SubDirCcFlags -DUSE_FAST_CODE ;
```

HEADER FILE PROCESSING

One of the functions of the *Object* rule is to scan source files for (C style) header file inclusions. To do so, it sets the **jam**-special variables $(HDRSCAN) and $(HDRRULE) as target-specific variables for the source file. The presence of these variables triggers a special mechanism in **jam** for scanning a file for header file inclusions and invoking a rule with the results of the scan. The $(HDRSCAN) variable is set to an **egrep**(1) pattern that matches "#include" statements in C source files, and the $(HDRRULE) variable is set to the name of the rule that gets invoked as such:

```
$(HDRRULE) source-file : included-files ;
```

This rule is supposed to set up the dependencies between the source file and the included files. The *Object* rule uses *HdrRule* to do the job. *HdrRule* itself expects another variable, $(HDRSEARCH), to be set to the list of directories where the included files can be found. *Object* does this as well, setting $(HDRSEARCH) to $(HDRS) and $(STDHDRS).

The header file scanning occurs during the "file binding" phase of **jam**, which means that the target-specific variables (for the source file) are in effect. To accomodate nested includes, one of the *HdrRule*'s jobs is to pass the target-specific values of $(HDRRULE), $(HDRSCAN), and $(HDRSEARCH) onto the included files, so that they will be scanned as well.

HdrRule Rule

HdrRule can be invoked directly, but it is most usable as the boilerplate in a user-defined $(HDRRULE). For example:

```
Main mkhdr : mkhdr.c ;
Main ugly : ugly.c ;

HDRRULE on ugly.c = BuiltHeaders ;

rule BuiltHeaders
{
        DEPENDS $(>) : mkhdr ;
        HdrRule $(<) : $(>) ;
}
```

This example just says that the files included by "ugly.c" are generated by the program "mkhdr", which can be built from "mkhdr.c". By calling *HdrRule* at the end of *BuiltHeaders*, all the gadgetry of *HdrRule* takes effect and it doesn't need to be duplicated.

Variables

The complete list of variables used by the *HdrRule* coterie are:

```
$(HDRPATTERN)     * scan pattern for $(HDRSCAN) (ugly egrep expression)
$(HDRRULE)          scan rule, when set activates scanning (HdrRule)
$(HDRS)           * non-standard directories for headers ()
$(HDRSCAN)          scan pattern when actually scanning ($(HDRPATTERN))
$(HDRSEARCH)        search list for HdrRule ($(HDRS) $(STDHDRS))
$(STDHDRS)        * standard directories for headers (/usr/include)
```

The *Object* rule sets HDRRULE and HDRSCAN specifically for the source files to be scanned, rather than globally. If they were set globally, **jam** would attempt to scan all files, even library archives and executables, for header file inclusions. That would be slow and probably not yield desirable results.

COPYING FILES

File Rule

The *File* rule copies one file to another. The target name needn't bear any relationship to the source name. For example:

```
File $(DESTDIR)/foo : bar ;
```

Bulk Rule

The *Bulk* rule is a shorthand for many invocations of the *File* rule when all files are going to the same directory. For example:

```
Bulk /usr/local/lib/grob : grobvals.txt grobvars.txt ;
```

HardLink Rule

The *HardLink* rule makes a hard link (using **ln**(1)) from the source to the target, if there isn't one already. For example:

```
HardLink config.h : config.h.dist ;
```

Shell Rule

The *Shell* rule is like the *File* rule, except that it makes sure the first line of the target is "#!/bin/sh" and sets the permission to make the file executable. For example:

```
Shell /usr/local/bin/add : add.sh ;
```

Variables

The following variables are used when copying files:

```
$(FILEMODE)       * default value for MODE for files (644)
$(SHELLHEADER)      first line of shell scripts (#!/bin/sh)
$(SHELLMODE)      * default value for MODE for shell scripts (755)
$(MODE)             permission on target
```

File and Shell sets a target-specific MODE to the current value of $(FILEMODE) or $(SHELLMODE), respectively. For example:

```
Shell /usr/local/bin/add : add.awk ;
SHELLHEADER on /usr/local/bin/add = "#!/bin/awk -f" ;
```

This installs an **awk**(1) script.

INSTALLING FILES

InstallBin Rule

InstallBin calls **install**(1) to install executables in the target directory. $(BINDIR) is set to /usr/local/bin for convenience. For example:

```
Main add : add.c ;
Main sub : sub.c ;
InstallBin $(BINDIR) : add sub ;
```

InstallLib Rule

InstallLib calls **install**(1) to install files in the target directory. $(LIBDIR) is set to /usr/local/lib for convenience. For example:

```
InstallLib $(LIBDIR) : bighelp.txt ;
```

InstallMan Rule

InstallMan calls **install**(1) to install manual pages in the appropriate subdirectories of the target directory. $(MANDIR) is set to /usr/local/man for convenience. For example:

```
InstallMan $(MANDIR) : add.1 sub.1 bigfile.8 ;
```

InstallShell Rule

InstallShell calls **install**(1) to install shell scripts in the target directory. For example:

```
Shell bugs : bugs.sh ;
InstallShell $(BINDIR) : bugs ;
```

The difference between *Shell* and *InstallShell* is not much: they both copy the source to the target. The former also makes sure the script begins with the magic string "#!/bin/sh"; the latter uses **install**(1) for the copy.

MkDir Rule

All the *Install* rules invoke the *MkDir* rule to create the directory for the target file. *MkDir* recursively invokes itself on its parent directory, to make sure the whole path gets created. *MkDir* marks directories with the built-in rule *NOUPDATE*, which tells **jam** not to update a target once it exists. In that way, the contents of the install directory can depend on the existence of the install directory itself, and thus the directory will be made before its contents are installed. You can call *MkDir* directly. For example:

```
File /usr/local/bin/junky : junky ;
DEPENDS /usr/local/bin/junky : /usr/local/bin ;
MkDir /usr/local/bin ;
```

This says that /usr/local/bin must be created before /usr/local/bin/junky can be built. Needless to say, **jam** can't do much if you don't have permissions to create directories along the path.

Variables

The following variables control the installation rules:

```
$(BINDIR)            InstallBin directory (/usr/local/bin)
$(LIBDIR)            InstallLib directory (/usr/local/lib)
$(MANDIR)            InstallMan directory (/usr/local/man)
$(INSTALL)           The install program; uses cp if not set (install)
$(FILEMODE)        * default MODE for InstallLib, InstallMan (644)
$(EXEMODE)         * default MODE for InstallBin (711)
$(SHELLMODE)       * default MODE for InstallShell (755)
$(MODE)              permission on target
$(MKDIR)             Program for creating a directory (mkdir)
```

The *Install* rules set a target-specific MODE to the current value of $(FILEMODE), $(EXEMODE), or $(SHELLMODE), depending on which *Install* rule was invoked.

The directory variables are just defined for convenience: they must be passed as the target to the appropriate *Install* rule. The $(INSTALL) and mode variables must be set (globally) before calling the *Install* rules in order to take effect.

HANDLING DIRECTORY TREES

Jam can build large projects spread across many directories in one pass, tracking the relationships among all files. It doesn't require the user to change the invocations of normal rules like *Main*, *Library*, etc. to use non-local pathnames: these rules continue to refer to files in the directory of the **Jamfile**. This section describes the rules and variables which support this.

To build a whole directory tree at a time, the user must do three things:

1. Set an environment variable pointing to the root directory of the source tree. The root variable's name is left up to the user, but in these examples we use TOP.
2. Place at the root of the tree a file named **Jamrules**. (This file can alternately be named by the variable $(xxxRULES), where xxx is the name of the root variable). This file could be empty, but in practice it contains user-provided rules and variable definitions that are shared throughout the tree. Examples of such definitions are library names, header directories, install directories, compiler flags, etc. This file is good candidate for automatic customizing with **autoconf**(GNU).
3. Preface the **Jamfile** in each directory with an invocation of the *SubDir* rule.

SubDir Rule

The *SubDir* rule does two things:

1. It reads in the **Jamrules** at the root of the tree, if that file hasn't already been read in by a previous invocation of *SubDir*.
2. It sets a few variables that tell **jam** the name of the **Jamfile**'s directory, so that **jam** may find source files that are named local to the **Jamfile**'s directory.

The *SubDir* rule takes as its first argument the root variable's name and takes as subsequent arguments the directory names leading from the root to the directory of the current **Jamfile**. Note that the name of the subdirectory is given as individual elements: the *SubDir* rule does not use system-specific directory name syntax.

The *SubDir* rule must be invoked before any rules that refer to the contents of the directory - it is best to put it at the top of each **Jamfile**. For example:

```
# Mondo src/util directory.

SubDir TOP src util ;

Main $(TOP)/bin/testutil : test.c ;

LinkLibraries $(TOP)/bin/testutil : $(TOP)/lib/libutil.a ;

Library $(TOP)/lib/libutil.a : gadgets.c gizmos.c widgets.c ;
```

This compiles four files in $(TOP)/src/util, archives three of the objects into libutil.a, and links the whole thing into $(TOP)/bin/testutil.

SubInclude Rule

The *SubInclude* rule sources the **Jamfile** from the named subdirectory. Its arguments are in the same format as *SubDir*'s, and its only reason for being is to allow including a subdirectory **Jamfile** without having to use system-specific directory name syntax.

The recommended practice is only to include one level of subdirectories at a time, and let the **Jamfile** in each subdirectory include its own subdirectories. This allows a user to sit in any arbitrary directory of the source tree and build that subtree. For example:

```
# Top level Jamfile for mondo project.
#
# $(TOP) points to root of mondo tree (set in environment).

SubInclude TOP src ;
SubInclude TOP man ;
SubInclude TOP misc ;
SubInclude TOP util ;
```

If a directory has both subdirectories of its own as well as files that need building, the *SubIncludes* should be either before the *SubDir* rule or be at the end of the **Jamfile** - *not* between the *SubDir* and other rule invocations. For example:

```
# Mondo src code.

SubDir TOP src ;

Main mondo : mondo.c ;
LinkLibraries mondo : $(TOP)/lib/libmisc.a $(TOP)/lib/libutil.a ;

SubInclude TOP src misc ;
SubInclude TOP src util ;
```

Variables

The following variables are used when **jam** spans multiple directories:

```
$(LOCATE_TARGET)   Directory to place targets.
$(SEARCH_SOURCE)   Directory to find sources.
$(SOURCE_GRIST)    Something to perturb source file names.
```

SubDir sets $(LOCATE_TARGET) and $(SEARCH_SOURCE) to be the directory given to *SubDir*. These variables are used extensively by rules in **Jambase**: most rules that generate targets (like *Main*, *Object*, etc.)

set $(LOCATE) to be $(LOCATE_TARGET) for the targets they generate, and rules that use sources (most all of them) set $(SEARCH) to be $(SEARCH_SOURCE) for the sources they use.

$(LOCATE) and $(SEARCH) are better explained in **jam**(1), but in brief they tell **jam** where to create new targets and where to find existing ones, respectively.

SubDir sets $(SOURCE_GRIST) to be a value derived from the directory name. $(SOURCE_GRIST) is used by the rules that take source files to perturb file names in different directories that would otherwise be the same.

It should be noted that the user can set these variables independently of *SubDir*, or after it. The most profitable example is setting $(LOCATE_TARGET) to be a directory outside the source tree: in this case, **jam** can build into a target directory (tree) without ever modifying the source tree.

VMS Notes

On VMS, the logical name table is not imported as is the environment on UNIX. To use the *SubDir* and related rules, you must set the value of the variable that names the root directory. For example:

```
TOP = USR_DISK:[JONES.SRC] ;

SubInclude TOP util ;
```

The variable must have a value that looks like a directory or device. If you choose, you can use a concealed logical. For example:

```
TOP = TOP: ;

SubInclude TOP util ;
```

The **:** at the end of TOP makes the value of $(TOP) look like a device name, which **jam** respects as a directory name and will use when trying to access files. TOP must then be defined from DCL:

```
$ define/job/translation=concealed TOP DK100:[USERS.JONES.SRC.]
```

Note three things: the concealed translation allows the logical to be used as a device name; the device name in the logical (here DK100) cannot itself be concealed logical (VMS rules, man); and the directory component of the definition must end in a period (more VMS rules).

MISCELLANEOUS RULES

Clean Rule

The *Clean* rule has only a simple action: to delete all of its sources. It is normally invoked with generated files as sources, so that they can be cleaned out. *Clean* must invoked with a target as well, as different sets of files might be cleaned on with different targets. To actually remove the files to be cleaned, you invoke **jam** with the target name on the command line. For example:

```
Clean zap : junk1 junk2 junk3 ;
```

Saying "`jam zap`" would cause it to delete junk1, junk2, and junk3.

All rules listed in this manual page that generate targets, except the *Install* rules, invoke the following *Clean* rule:

```
Clean clean : $(<) ;
```

The *Install* rules invoke the following:

```
Clean uninstall : $(<) ;
```

Thus a "`jam uninstall`" removes anything created with the *Install* rules, and a "`jam clean`" removes anything created by the other rules listed in this manual page. It should be noted that **jam**'s cleaning mechanism gets rid of exactly the files it created, not miscellaneous junk left around by the user.

Given user-defined targets, the *Clean* rule can selectively remove other generated files. For example:

```
rule M4
{
```

```
                # File depends on it's m4 source

                DEPENDS $(<) : $(>) ;

                Clean m4clean : $(<) ;
        }

        actions M4
        {
                m4 < $(>) > $(<)
        }
```

Here a "`jam m4clean`" would remove all files created by **m4**.

RmTemps Rule

Some intermediate files are meant to be temporary. The *RmTemps* rule marks such files with the *TEMPORARY* rule, and then deletes them after they are used. To delete them only when they are finished being used, *RmTemps* must be the last rule (with actions) invoked on the target that uses the temporary files, and the sources to *RmTempts* must be the temporary files themselves. For example:

```
SpecialUserRuleA foo : bar ;
SpecialUserRuleB ola : foo ;
RmTemps ola : foo ;
```

This says: build "foo" using *SpecialUserRuleA* and "ola" using *SpecialUserRuleB*. Once that is done, remove "foo".

SPECIAL TARGETS

Jam has only one special target: *all*, which it tries to build if no targets are on the command line. **Jambase** defines several special targets which are dependencies of *all*:

```
all             - parent of first, shell, files, lib, exe
first           - first dependency of 'all', for potential initialization
shell           - parent of all Shell targets
files           - parent of all File targets
lib             - parent of all Library targets
exe             - parent of all Main target
dirs            - parent of all MkDir targets
clean           - removes all Shell, File, Library, and Main targets
uninstall       - removes all Install targets
```

Jambase marks all of these targets with **jam**'s *NOTFILE* attribute, meaning that they aren't to be found in the filesystem. You can build selected components by giving *shell*, *files*, *lib*, *exe*, or *dirs* as targets on the command line. You can remove the files that **jam** built giving *clean* and *uninstall* as targets. And you can arrange for **jam** to run initialization commands by putting actions on the target *first*. For example:

```
actions Initialize
{
      ECHO "This is a test of the jam initialization system."
}

Initialize first ;
```

This only gets run if **jam** is invoked with the *all* or *first* targets, or no target at all.

JAM BUILT-IN RULES AND VARIABLES

This section discusses **jam**'s built-in rules and variables. They are described in **jam**(1) more precisely. Built-in rules are uppercase, as opposed to the mixed-case rules defined by **Jambase**. These built-in rules, along with the other **jam** syntax for manipulating variables, provide the foundation upon which the **Jambase** is built. A **Jamfile**, or (more likely) a **Jamrules** (q.v.), can make use of these built-in rules and variables as well.

DEPENDS, INCLUDES Rules

Two rules build the dependency graph. *DEPENDS* simply makes its sources dependencies of its targets. *INCLUDES* makes its sources dependencies of anything of which its targets are dependencies. 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 ;
```

Both "foo.c" and "foo.h" become dependencies of "foo.o" in this example.

ALWAYS, LEAVES, NOCARE, NOTFILE, NOUPDATE, TEMPORARY Rules

Six rules mark targets so that **jam** treats them differently during its target binding and updating phase. Normally, **jam** 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 with the target name as the rule's target:

The *ALWAYS* rule causes its targets to be always updated. 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.

The *NOCARE* rule causes **jam** to ignore its targets if they can't be found and have no updating actions. Normally, **jam** issues a warning about a target that can't be built and then refuses to build anything that depends on that target. The *HdrRule* uses *NOCARE* on the header file names found during header file scanning, to let **jam** know that the included files may not exist. For example, if a #include is within an #ifdef, the included file may not actually be around.

The *NOTFILE* rule marks its targets as being pseudo targets, that is, targets that aren't really files. 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 *all* and *clean* targets are examples of such targets.

The *NOUPDATE* rule causes **jam** to ignore the modification time of the target. 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. 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.

The *TEMPORARY* rule allows for targets to be deleted after they are generated. If **jam** sees that a temporary target is missing, it will use the target's parent's time when determining if the target needs updating. Object files that are also archived in a library are marked as such, so that they can be deleted after they are archived.

The *LEAVES* rule makes each of the targets depend only on its "leaf" dependencies. 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.

ECHO, EXIT Rules

These two rules help during the **Jamfile** compiling phase. The *ECHO* rule just echoes its targets to the standard output. The *EXIT* rule does the same and then does a brutal, fatal exit of **jam**.

SEARCH, LOCATE Variables

These two variables control the binding of target names to real files: they indicate what path name is to be prepended to the target name to get to the real file. $(SEARCH) provides a list of directories along which **jam** scans looking for a target. $(LOCATE) overrides $(SEARCH), indicating the directory where the target must be. Normally, $(SEARCH) is set for existing targets while $(LOCATE) is set for the targets which **jam** must build. If neither $(SEARCH) nor $(LOCATE) are set, or if the name of the target is a rooted file name (i.e. on UNIX beginning with "/"), then the file name is assumed to be the target name.

Both $(SEARCH) and $(LOCATE) should be set target-specific and not globally. If they were set globally, **jam** would use them for all file binding - including looking for the *Jamfile*, and this is not likely to produce sane results. All of the rules defined in **Jambase** (and described in this document) set $(SEARCH) and $(LOCATE) to sensible values for sources they are looking for and targets they create, respectively. These values are usually $(SEARCH_SOURCE) and $(LOCATE_TARGET), described above in the section describing variables use when building whole directory trees. The header file processing rule *HdrRule* sets $(SEARCH) for header files to be $(HDRS).

When writing your own rules, especially ones not built upon those in **Jambase**, you may need to set $(SEARCH) or $(LOCATE) directly. Most often you'll set them to the prevailing value of $(SEARCH_SOURCE) or $(LOCATE_TARGET). The best examples are those in **Jambase**.

HDRSCAN, HDRRULE Variables

These two variable control header file scanning. The first is an **egrep**(1) pattern, with ()'s surrounding the file name, used to find file inclusion statements in source files. The second 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 **jam** 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. That is, there is no need to run something like **makedepend**(GNU) to create a static dependency file. Because **jam** uses regular expressions to find include files, it can't know when an include is within #ifdefs or other conditional logic. To make up for this, *HdrRule* applies the *NOCARE* rule to each header file, just in case it is bogus. Also, regular expressions only work where the included file name is literally in the source file. They can't handle languages that allow including files using variable names (as **jam**'s own langauge does).

JAMSHELL Variable (Unix Only)

When **jam** 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 is controlled by $(JAMSHELL), whose default value is:

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

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

On UNIX **jam** can build targets in parallel, as long as the dependencies among files are properly spelled out and actions don't create fixed named files in the current directory. (If either of those two provisions are violated, **jam** can trip over itself when building in parallel things which just happen to build OK sequentially.) When building in parallel, **jam** simply forks off more than one shell at a time.

Jam 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 **jam** 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
```

SEE ALSO

jam(1), **Jambase**(5)

BUGS

This document shouldn't be a manual page.