BUILD: a Tool for Program Building

Douglas S. Rand

PE-T-1283, Rev 5

October 9, 1987

Date: October 9, 1987
To: R & D developers
From: Douglas S. Rand
Subject: BUILD: a Tool for Program Building
Reference: PE-T-1346 RCS: Revision Control System
Keywords: Unix, make, project build, configuration management

Abstract

In building products and projects we often find the need to change several modules and then rebuild the project. Recompiling individual modules leads to stale binaries and full rebuilds are costly in terms of CPU usage. An answer to this is a controlled rebuild of the project where all the components are brought up to date. BUILD is a tool which takes a dependency description and selectively brings a project up to date. It is similar to Unix MAKE in concept and execution. BUILD will run on 19.4 and later systems.

New in revision (22.0.7 and beyond) is support for RCS archives and efficiency additions. New in this revision (22.0.16 and beyond) is improved rule handling. Changes from 22.0.7 and 22.0.16 have change bars.

1. Introduction

In maintaining products we typically deal with a multitude of source, binary and include file making up the total. As a single source or include file changes we are faced with the option of attempting to selectively recompile these files or doing a wholesale rebuild of the product. BUILD answers this need by allowing the user to describe the interdependencies between these files and the finished product.

In describing these dependencies we will talk of *components*, which are usually files, and *dependencies* which are always treated as components.

BUILD makes assumptions. The initial assumption is that a binary file component implies a source file. A simple build file will serve to illustrate:

```
foo.run: foo.bin bar.bin
   bind -lo foo bar -li -file foo.run
```

In this example foo.run (the program EPF) is dependent on foo.bin and bar.bin. BUILD will also infer (let's assume foo.spl and bar.spl exist) that foo.bin is dependent on foo.spl and bar.bin is dependent on bar.spl. When BUILD uses this file it will walk through these dependencies *depth then breadth*. The search looks like this:

foo.run
- foo.bin
  - foo.spl
- bar.bin
  - bar.spl

As BUILD walks this network it gets the date-time modified from the file system and as it passes up the network it returns this value. When it finds a component that has a dependency that is newer then the component's date-time modified it tries to bring that component up to date. If this succeeds then the current date-time is returned up the network. So if foo.spl is newer then foo.bin BUILD will recompile foo.spl and it will also bind foo.bin and bar.bin to create a new copy of foo.run.

BUILD uses a database of dependencies and commands to update a product when a dependent source or include file changes. No longer does a developer have to worry day by day which sources are dependent on which include file and how to bring the product as a whole up to date. He only has to maintain a central description in the build file as the source changes.

2. Command Description

Capital letters in a flag are required. So -No_commands can be abbreviated on the command line to -N.

BUILD [component]
[-No_commands]
[-DeBuG]
[-From pathname]
[-Ignore_errors]
[-Verbose]
[-Keep_tempfiles]
[-SDI]
[var1=value1 ..]
[-Help]

There are no required arguments and the binary flags -No_commands and -DEBUG default to off.

component	The part of the dependency net that should be built. This defaults to the first dependency rule read in.
-No_commands	Tells BUILD to print what commands will be executed without actually doing them.
@CB[-Verbose	Puts out warnings for various non-serious conditions. Useful when debugging a buildfile or when simple problems occur. The -Debug option will imply -Verbose but puts out much more output.]
-DEBUG	Abbreviated -DBG. Causes alot of debugging information to be printed. If you have a problem then it would be very useful to have a como file of BUILD rerun with the -DEBUG flag. See also the section on meta commands to get a shorter debug output.
-From	This is the pathname for the dependency file. The default is BUILDFILE in the current directory (or BUILDFILE.BUILD). The suffix .BUILD is understood so for your product.BUILD file you can simply specify -f product.
-Ignore_errors	Don't stop building if an error is found. BUILD keeps count of errors and warnings if this flag or -SDI (which turns on this flag as well) are specified.
-Keep_tempfiles	If you use CPL code then BUILD generates temporary files for execution. This option leaves them for your perusal or debug purposes.
-SDI	This must be used for SDI submittals. This forces all commands and compilations to occur (as if everything is out of date) and implies the -Ignore_errors option. @cb[This sets the local BUILD variable *SDI* to 'true'.]
var1=value1..	There may be any number of var=value pairs on the line to set initial variable values (this is the way to pass arguments). There may be no whitespace in this, i.e. foo=bar is o.k. whereas foo = bar and bletch is not. @cb[After rev 22.0.7 whitespace is OK. Also foo = 'bar and bletch' will do the intuitively correct thing by using quotes to enclose *bar and bletch.*]

3. An Example

I will use the build file for BUILD as an example to start with.

```
/* BUILD.BUILD, BUILDSRC, ENVIRONMENTS, 04/08/86
/* buildfile for build
/* Copyright (c) 1986, Prime Computer, Inc., Natick, MA 01760 */
/*                     All Rights Reserved */
/*
/* List of object files (note the \ which indicates a line
/* continuation)
objs = build.bin reader.bin utilities.bin dtm.bin\
  uptodate.bin expand_line.bin path.bin

/* List of include files
insert = utilities.ins.spl make_ds.ins.spl

/* To compile an spl module
spl -> bin
  spl {*} -b *>object>{=}.bin

/* To make build_product.run bind all the objects and rename
/* the result build_product.run.  build_product.run is dependent on
/* all the objects and include files
build_product.run: {objs} {insert}
  bind -lo {objs} -li -dynt -all -ng
  cname build.run build_product.run
/* build.bin ... are each dependent on all the include
/* files (and their sources implicitly)
build.bin: {insert}
path.bin: {insert}
reader.bin: {insert}
utilities.bin: {insert}
dtm.bin: {insert}
uptodate.bin: {insert}
expand_line.bin: {insert}

/* To clean up the binaries one would type build_product clean
clean:
  delete *>object>@.bin -nq -nvfy
```

4. General Information

4.1 Introduction

BUILD is a tool which takes dependencies and rules and uses them to build a product. The dependencies form a network and the component argument tells BUILD which network component to start at. Rules and commands attached to the dependencies are used to bring each component up to date.

BUILD is a qualified tool and may be used for SDI submittals for rev 21.0 and later. Build files must have a .BUILD suffix and a standard master disk header.

4.2 Components

Components are usually simply filenames and sometimes they are placeholders.

Binary — Binary files are assumed if the name of the component ends in ".bin". BINARY$ search rules are used to find the date-time last modified of the file and a source is presumed to exist and to be a dependency. The source is obtained using the SOURCE$ search rules and the contents of the SUFFIXES variable. If no commands are given then either a rule or a default rule will be used to bring this component up to date.

Source — No assumptions are made insofar as dependencies. The date-time last modified is found using the SOURCE$ search rule. Rules or default rules may be used to build the source file. The 'to' suffix is obtained from the first dependency (ordering will be important here).

Placeholder — This is simply a non-suffixed component name. It is assumed not to be buildable using a rule. This component will always execute commands if it has them. It is given a default date-time modified of 1964.

Include — Include files may depend on other include files. Commands will be executed if given and the component is not up to date. Otherwise these components act like source files.

Modula Definition files — Special handling is given to files that end in .def.mod. The file is found using the DEFINITION$ search rules. The 'to' suffix is set to 'def.mod' for the purpose of rules.

Each component is brought up to date depending on whether the component's starting date-time modified (as described by the file system or the default) is less then the latest changed dependency (not up to date). Since dependencies are components one may consider that a component may pass it's dtm (date-time modified) back to another component as that other component's dependency. If a component has no dependencies then it is, by default, considered to be automatically up to date (that is leaf nodes in this network are always up to date).

It is important to note that once a component has been checked and brought up to date it is not rechecked. Ordering of your dependencies and components may be important.

@CB[RCS archives have dtm information per revision encoded in the file. See the chapter on RCS for information on using RCS archives with BUILD.]

4.3 Search rules

When BUILD looks for a file it will use one of three search rules. If the file is a binary file then it will use the BINARY$ search rule. If the file is an include file (either contains .ins. in the entryname or a suffix of .h) then it uses the INCLUDE$ search rule. For MODULA definition modules (.def.mod) it uses the DEFINITION$ search rule. For all other files it uses the SOURCE$ search rule. If one or more search rules are undefined it will be the equivalent of using the current directory.

Builtin search rules can be overridden. See the section on meta directives.

5. File Description

There are five kinds of structure in the build file:

- Comment lines
- Variable assignments
- Rules
- Dependancies
- Meta directives

Comment lines begin in column one with a '/*'. The remainder of a comment line is ignored.

5.1 Meta directives

If a line starts with a '%' then it is a meta line and there are the following options:

%include filename

This starts reading in *filename*. *Filename* may also include a *%include* directive. When BUILD is finished reading the file in the %include it will continue reading the original file which contained the %include. This is not usable while processing commands for a rule or node.

%use search_rule_name for suffix list

This allows the user to override the default search rule handling. *Suffix list* is a blank separated list of suffixes, i.e. 'c spl mod'. Search rule names are normally capitalized. The suffix may also be ins for all ..ins.. files or def.mod for definition module files.

%command primos command line

This allows for execution of PRIMOS commands at read time. This is particularly useful for setting search rules.

@cbon()

%revision revision [node]+

This sets the revision for a particular node or nodes. This is used in conjunction with RCS archives. This is a useful feature for determining the configuration of a particular build since one normally wants all the most current sources but for a particular source one may want an experimental version or an older version. %revision may be abbreviated %rev as well.

The revision given may be either an explicit revision such as 3.2 or a named RCS revision such as set by rcs -n*name* archive_file.

Remember that the nodes named should be RCS archives and **not** source filenames. **%revision 19.4 foo.spl_v bar.plp_v** is a valid example of this feature. @cboff()

%debug on/off

Essentially the same as -debug command line option but lets you control what portion of the source you actually want debug info for.

%echo on/off

Echo line expansions. Useful for debugging problems with variable expansions and quoting.

%if,%then,%else,%end

An %if line introduces a conditionally read section. The sections are delimeted by %then, %else, and %end. The following illustrates this:

```
%if {DEBUG_VAR}
%then
COMPILE_FLAGS = -debug
%end
%if [calc {count} = 5
%then
...
%else
...
%end
```

The %then can be ommited. See the section on line expansion for more details on using PRIMOS command functions from BUILD.

The expression is considered 'false' if the text is:

'' <- meaning blank
F
FALSE
f
false
0

@CBon()

%source_lookup on/off

If turned off then binary files are treated as all other files and will not trigger automatic lookup of their source files. Also the suffixes variable is effectively ignored. @CBoff()

5.2 Line expansion and reading

All non-comment, non-blank, non-command lines which are read in are passed through an expander which expands any variables in the line and which removes one level of quoting. For example:

```
line = {foo} is a good "way of expressing {foo}"
```

will result in a line read in (assume the value of foo is "once upon a time"):

```
line = once upon a time is a good way of expressing {foo}
```

Quotes may be expressed inside of quotes by using a double quote (""). So:

```
"a "" b """ => a " b "
```

Another important note is that lines ending in '\' will be continued on the next line. This is useful for both dependencies and variable. This is **not implemented** for commands.

The '\' character is also used as a one character quote. That is \{ is equivalent to "{" and \\ is equivalent to "\".

Also the line is scanned for '['. If a '[' is found then a matching ']' is search for and the contents between the left and right delimeters is passed to PRIMOS as an active function. So:

```
date = [date -full]
```

will set the *date* variable to the current date. Another example:

```
%if [calc [date -tag] > 860101]
%then
options = -newfortran {options}
%end
```

will add -newfortran to the options if the current date is after 1/1/86.

5.3 Command execution

When commands are executed they are passed through the expander and the results passed to the system subroutine CP$. This allows one to execute any Primos command or CPL. If any command line has a '&' character in it then the entire set of command lines (either bound to a component or a rule) is passed to CPL in a temporary file.

5.4 Variables

Variable assignments are of the form variable = value. There is only one of these per line. The value may be anything and include other variables or quoted areas. A variable is defined as any set of printable characters, e.g. foo is a variable name, so is 2late. If a variable name starts with a '.' it is considered a Primos global variable and has the restrictions for that class. Either type of variable may be set in the file or on the command line. References to variables is similar to CPL. They are quoted with '{}'. Expansion may be inhibited by using double quotes, one level of quotes will be removed. Examples:

```
/* set foo to the value trivial pursuit
foo = trivial pursuit
/* set bar to the value what a game
bar = what a game
/* statement will have the value:
/* trivial pursuit : what a game
statement = {foo} : {bar}
/* the Primos global variable .trivia will also
/* have the value of statement
.trivia = {statement}
/* This gives bletch the value: {foo} : what a game
bletch = "{foo}" : what a game
```

Predefined special variables are:

*	See description under RULES.
=	See description under RULES.
-	See description under RULES.
SUFFIXES	Contains the list of source suffixes searched for (in order). The default list is 'cc spl mod pl1 pl1g plp pascal ftn f77 cbl vrpg pma.' (PL1,PL1G added at 21.0.7)
@cb[REVISION	Can be empty, 'latest', or one, two or three dot separated numbers or a named RCS revision. (e.g. 1, 1.1, 2.4.2) See the chapter on using RCS from BUILD for details.
SDI	This is set to 'true' if the -SDI command line flag is set.]

5.5 Rules

Rules explain how to create one kind of file from another. File kinds are determined by suffixes. BUILD understands (counts upon) our meanings for the suffixes .bin, .h (c header files), and ..ins.. (conventional include files). All other suffixes are assumed to belong to the category of sources. A rule is expressed:

```
suffix_from -> suffix_to
   command1
   command2
   ...
   commandn
```

A rule should have a minimum of one command. In writing rules one may assume that three special local variables have values. They are:

{*}	The full pathname of the source file.
{=}	The entryname of the file (no suffix).
{-}	The name of the dependency which triggered the update.

For example a rule to compile a SPL module:

```
spl -> bin
   spl {*} -b myobjdir>{=}.bin {splopts}
```

BUILD used to default if it searches for a rule and none can be found. @cbon() There are builtin rules for all standard PRIME languages (and other random things). The current list is:

```
spl -> bin
  spl {*} {splflags}
f77 -> bin
  f77 {*} {f77flags}
ftn -> bin
  ftn {*} {ftnflags}
pl1 -> bin
  pl1 {*} {pl1flags}
pl1g -> bin
  pl1g {*} {pl1gflags}
c -> bin
  cc {*} {cflags}
cc -> bin
  cc {*} {cflags}
pascal -> bin
  pascal {*} {pascalflags}
plp -> bin
  plp {*} {plpflags}
pma -> bin
  pma {*} {pmaflags}
mod -> bin
  modula {*} {modflags}
def.mod -> sym
  modula {*} {defmodflags}
cbl -> bin
  cbl {*} {cblflags}
deremer -> c
  deremer {*} -cc {deremerflags}
deremer -> cc
  deremer {*} -cc {deremerflags}
deremer -> spl
  deremer {*} -spl {deremerflags}
deremer -> plp
```

deremer {*} {deremerflags}
vrpg -> bin
vrpg {*} {vrpgflags}

@cboff If from = to then nothing is done.

5.6 Dependancies

Dependancies explain how some components rely on other components. A component will either be a file or a placeholder as explained under the section on components. The basic format is:

```
component: dependency1 .. depn
   command1
   command2
   ...
   commandn
```

Commands are optional. If there are no commands then BUILD will look for a rule to create the component. If the component is a binary file then BUILD will automatically assume that it's source file is a dependent, i.e. if the component is foo.bin then you needn't specify foo.spl. BUILD uses the SUFFIXES variable to determine which sources suffixes to search for and in what order. If the default list is wrong the you may change it in the usual fashion by including a variable definition line for SUFFIXES.

Some special components exist.

<begin>	is a component that is brought up to date before the main component.
<end>	is a component that is brought up to date after the main node.
@CBon() <error>	has it's commands executed on error (appropriate when using -SDI or -Ignore_errors).
<warning>	has it's commands executed on warnings (appropriate as <error>). @CBoff()

6. Using RCS archives

6.1 Introduction

RCS archives are single files that contain the current revision of a file and backward deltas to older revisions of the text. (See PET-1346 for a full description of RCS) Branches are handled as 'forward' deltas from the main trunk.

Using the file system dtm for an RCS archive is clearly not the correct way to determine the 'datedness' of an archive file. BUILD has some extensions to deal with this properly.

The *revision* variable determines the behaviour of BUILD when it encounters a RCS archive. Any file whose name ends in '_V' is assumed to be an archive file. Rules can be constructed in the normal fashion. There are no builtin rules to handle RCS files.

If the *revision* variable is set to '' or 'latest' then the first revision listed in the archive file (the most recent) is taken as the dtm. If the *revision* variable has one or two digits (a trunk revision) then the first match searching forward is used. If the *revision* variable has three digits then a branch rev is searched for and the latest branch rev (highest fourth number) is used. If the *revision* variable has four digits then the first exact match is used.

The rules for resolving *revision* also apply to specific revisions assigned with %revision.

Named revision are mapped on a per archive file basis to actual revisions in each archive file. This provides a good mechanism for maintaining configurations. (See *rcs -n* in PE-T-1346)

6.2 Rules for RCS archives

The include files should be brought up to date first. This means that if the version in the file is not up to date compared to the archive then the later copy should be checked out. Conversely one may wish to always extract the correct revision. Perhaps the safest way has the file dtm for revisions when the latest revision is used and always extracts otherwise. The rule for extracting include files would probably look like this:

```
ins.spl_v -> ins.spl
   rco -p *>insert>{=}.ins.spl
```

For source files the file need only be extracted if the binary is out of date. The following is recommended as a typical rule:

```
spl_v -> bin
   rco -p *>insert>{=}.spl
   spl *>source>{=}.spl -b *>object>{=}.bin
```

This will also leave browsing copies of source files about.

The suffix list needs also to be modified for source lookup:

```
SUFFIXES = spl_v cc_v
```

In RCS revision may also be named. Using the %revision meta directive one may set the required revision explicitly for named files. Using named revisions will probably be an easier way to maintain older configurations of software then trying to explicitly name the numerical revision for each archive file.

7. Writing and debugging BUILDFILES

7.1 Writing BUILDFILES

There are several steps in creating a buildfile. The simplest buildfiles use defaults entirely. For example

```
my_project.run: my_project.bin my_util_file.bin
   bind -lo my_project my_util_file -li
```

is entirely sufficient to create a program my_project. This is a fine use for BUILD but most programs are more complex.

The first step is to get a list of binary files and find the include file dependencies on the corresponding sources. If the project is a large one then the resultant file may want to be separate from the main buildfile. So the main buildfile then has a line '%include auxilliary'. This makes BUILD include the file *auxilliary* or *auxilliary.BUILD* (files with .BUILD are selected first) as if the text in the auxilliary file was in the main file. Each node may have more then one dependency line.

```
foo.bin: x.ins.spl
foo.bin: y.ins.spl
```

will do exactly the same thing as:

```
foo.bin: x.ins.spl y.ins.spl
```

Important to note is that only one set of commands will be accepted. If two dependency lines both have commands then the last dependency line read will take precendence.

A good way to organize is to have dependencies in this aux file and the commands that are specific to individual modules (remember that the general cases can be expressed as rules) are given in the main file. A simple case of this could be:

```
/* FOO.BUILD

/* blank options normally are set to things like -debug or extended
/* optimization
options =

/* No sense in looking for cc,  plp,  etc. since this only uses spl
suffixes = spl

spl -> bin
  spl *>source>{=} -b *>object>{=}.bin {options}

foo.run: foo.bin bar.bin bletch.bin
  bind -lo foo bar bletch -li

%include aux

/* bletch always is compiled in debug mode
bletch.bin:
  spl *>source>bletch.spl -b *>object>bletch.bin -debug
```

```
/* AUX.BUILD general dependencies
foo.bin: x.ins.spl y.ins.spl
bar.bin: z.ins.spl
bletch.bin: x.ins.spl
```

While the names are trite the point is to split the stuff which doesn't change often (the basic build) and the stuff

which changes with the source (the dependencies). This also makes using tools which automatically generate the dependency file much less painful.

The only way that this becomes more complicated is that one can have include files dependent on other include files or multilayered dependencies with deremer. Some more examples (snippets of the files):

```
/* Rule for translating deremer to spl
deremer -> spl
   deremer {*} -spl

foo.bin:
/* foo.spl is dependent on foo.deremer
foo.spl: foo.deremer
```

Another example:

```
foo.bin: x.ins.spl
/* x.ins.spl includes y and z
x.ins.spl: y.ins.spl z.ins.spl
```

7.2 Common Problems

7.2.1 Search rules

The biggest problem by far is screwed up search rules so that BUILD doesn't find the files referenced. This is really a very simple minded tool. The easy way (after 22.0.7) to find this problem is to turn on -verbose and listen to the warnings that source files aren't found and that files aren't found. Sometimes this doesn't matter. For example: you have a node that is a placeholder. A placeholder won't be found in the file system. A placeholder's commands are always executed. If something else, like a binary file, isn't found then the usual culprit are the search rules.

7.2.2 Updating components

A second problem is that BUILD doesn't seem to be updating things that it should. This is often caused by not finding referenced files but also is caused by the simple lookup used. If the node name doesn't match the dependency name (like one of them is foo>bar.spl and the other is bar.spl) then no matter how obvious the association BUILD will not find it. The names must match letter for letter.

7.2.3 Other problems and bug reports

Other problems occur. If a problem occurs and -verbose doesn't help then try either -debug or %debug on and off to find the problem. You will get alot of output. Look to see that file lookups use the right search rules and that induced rules are correct. Sometimes problems occur with the preset variables *, = and -. These settings are displayed when debug output is on.

When everything fails or you think you've found a bug in BUILD then please send me:

- The buildfile(s)
- A como file of the -debug output (all of it please).

This should be sent either to x.mail doug -on enb or pdnmail doug@enx.

Table of Contents