Ebuild

COLLABORATORS			
	TITLE : Ebuild		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 7, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Ebuild

1.1 Ebuild.guide

```
                    This file documents version 0.97 of the Ebuild tool.  You may ↩
                       find
in this document:


                  Introduction
                      An introduction to automatic program building

                  Invoking EBuild
                      EBuild command-line arguments

                  Build Files
                      Buildfiles tell EBuild what to do.

                  Misc
                      What Build does and what it's good for

                  Bugs - Future
                      Known and removed bug and the future of EBuild

                  History
                      The past of EBuild

                  The Authors
                      Who did it?
```

1.2 Ebuild.guide/Introduction

```
Introduction
************

   EBuild is a Make clone, and it functions likewise. It is a tool that
```

helps you in recompiling necessary parts of a large application after modification.

You write a file .build in the directory that contains the sources of your project. The file contains info about which sources depend on which, and what actions need to be performed if a module or exe needs to be rebuilt.

EBuild checks the dates of the files to see if a source has been modified after the last compilation, and if the source uses modules that also have been modified, it will compile these first.

1.3 Ebuild.guide/Invoking EBuild

Invoking EBuild
****

EBuild can be run from any shell. If run without any arguments it reads the file .build and performs the actions of the first target in this file if any of the dependancies is newer.

The arguments are:

TARGET,FROM/K,FORCE/S,VERBOSE/S,NOHEAD/S,CONSTANTS/S,MESS/S:

TARGET
Build the provided target rather than the first in the build file.

FROM
Allows you to use another file than .build

FORCE
Rebuild, regardless of whether it was really necessary.

VERBOSE
Print the actions while executing them.

NOHEAD
Don't print the heading line (version and copyright).

CONSTANTS
Print all symbolic constants and the strings they are replaced with.

MESS
Don't delete the action script after execution. Useful for debugging. Note that the script file name is target dependant.

1.4 Ebuild.guide/Build Files

```
                Build Files
***********


                 Symbolic Constants

                 Including Files
                   Build files are normally named .build. This is the file EBuild ↩
                       looks
for when it is run.

   The syntax of build files equals that of unix-make. In general, #
precedes lines with comments, and:

     target: dep1 dep2 ...
       action1
       action2
       ...

   target is the resulting file we're talking about, in most cases an
exe or module, but may be anything. Following the : you write all files
that it depends upon, most notably its source, and other modules.

   The actions on the following lines are normal AmigaDos commands, and
need to be preceded by at least one space or tab to distinquish them
from targets.

     bla: bla.e defs.m
          ec bla quiet

   This simple example will only recompile bla.e if it was modified, or
if the defs.m which it uses was modified.

   If you type build with no args, build will ensure the first target
in the file to be up to date.

   A longer example:

     # test build file

     all:    bla burp

     defs.m: defs.e
             ec defs quiet

     bla:    bla.e defs.m
             ec bla quiet

     burp:   burp.e
             ec burp quiet

     clean:
             delete  defs.m bla burp

   This build file is about two programs, bla and burp, of which bla
```

```
also depends on a module defs.m. An extra target clean has been added
so you can type build clean to delete all results. The clean target is
called a phony target or fake target since it's not a real file.

   The all target is a fake target, too. It's okay to have multiple
fake targets, however, these cannot be used as dependancies.

   Other dependencies and actions are easily added. For example, if
your project uses a parser generated by E-Yacc:

     yyparse.m: parser.y
                eyacc parser.y
                ec yyparse quiet

   Or incorporates macro-assembly code as often used tool module:

     blerk.m: blerk.s
              a68k blerk.s
              o2m blerk
              copy blerk.m emodules:tools
              flushcache tools/blerk
```

1.5 Ebuild.guide/Symbolic Constants

```
                Symbolic Constants
==================

   In EBuild a symbolic constant is a string bound to a name.  Those
symbols can be used in rules and actions. The string of a symbol will
be inserted wherever  the symbol is found. Example:

     options=IGNORECACHE LINEDEBUG DEBUG
     test:   test.e
             ec test.e $(options)
             ==> ec test.e IGNORECACHE LINEDEBUG DEBUG

   The following example shows how to use constants in rules:

     testfile=bla
     $(testfile):    $(testfile).e
                     ec $(testfile).e

   There are two special symbols in EBuild. The first, target, holds
the name of the target the current action belongs to. dep gets the name
of the first dependancy of the current target.

   All except these two preset symbols may be used in rules as well as
in actions. target and dep, however, may only be used in actions. It's
safe to have it in rules, EBuild just aborts with a message that tells
you that it doesn't know this symbol.
```

```
                target

                dep

                Example for Constants
```

1.6 Ebuild.guide/target

```
target
------

   In the example below we tell EC to compile the target instead of
writing the actual name:

     options=IGNORECACHE LINEDEBUG DEBUG
     test:   test.e
             ec $(target).e $(options)

   This may seem to be not too useful, but take a look at this example:

     options=IGNORECACHE LINEDEBUG DEBUG
     test:   test.e
             ec $(target) $(options)
             if warn
             echo "Error: compile failed"
             else
             echo "Compiled OK... running"
             $(target)
             endif

   It's largely equivalent to the old code below, but allows more.

     options=IGNORECACHE LINEDEBUG DEBUG
     all:    test
             echo "ok, running:"
             test

     test:   test.e
             ec -q test $(options)
```

1.7 Ebuild.guide/dep

```
dep
---

   Another preset symbol is dep. It holds the name of the first
dependancy of the current target.

   Again, look at this example from the introduction of symbolic
constants:
```

```
     test:   test.e
             ec test.e IGNORECACHE LINEDEBUG DEBUG
```

Using dep would give this fragment:

```
     test:   test.e
      ec $(dep) IGNORECACHE LINEDEBUG DEBUG
```

1.8 Ebuild.guide/Example for Constants

```
Example for Constants
---------------------
```

This example is the one from an earlier section. We will use symbols to generalize it.

```
     options=IGNORECACHE LINEDEBUG DEBUG
     test:   test.e
             ec test IGNORECACHE LINEDEBUG DEBUG
             if warn
             echo "Error: compile failed"
             else
             echo "Compiled OK... running"
             test
             endif
```

As a first step every use of the actual name test is replaced by a constant and the compiler options are put in a symbol:

```
     OPTIONS=IGNORECACHE LINEDEBUG DEBUG
     PGM=test
     $(PGM): $(PGM).e
             ec $(pgm) $(OPTIONS)
             if warn
             echo "Error: compile failed"
             else
             echo "Compiled OK... running"
             $(PGM)
             endif
```

To indicate that we want to handle the target of the actions we should rather use $(target) instead of the $(PGM). Note that both uses are correct. We should use $(dep), too.

```
     PGM=test
     OPTIONS=IGNORECACHE LINEDEBUG DEBUG
     $(PGM): $(PGM).e
             ec $(dep) $(OPTIONS)
             if warn
             echo "Error: compile failed"
             else
             echo "Compiled OK... running"
             $(target)
```

```
            endif
```

Take a look at the line where the source is compiled. This line can be used for every source since the name of the file is in the preset symbol. If you like you could even make a symbol that holds this line:

```
COMPILE=ec $(dep) $(options)
```

1.9 Ebuild.guide/Including Files

```
Including Files
===============
```

When a file is included its contents are copied in the current build file before it is processed. The copying is only temporary, both the build file and the included file are left untouched.

Including is useful if you have a number of build files that all need the same variables or share some fake targets.

To include a file the build file has to have the '#i' directive followed by the name of the file to include. For example,

```
#i /scripts/template
```

includes the file /scripts/template in the current build file. Let's say this file contains these symbol definitions:

```
COMPILER=E:bin/EC
TEMP_DIR=T:
```

The following build file includes these definitions and uses it:

```
#i /scripts/template

test: test.e
  $(COMPILER) $(dep)
  Copy $target TO $(TEMP_DIR)
```

EBuild takes everything it finds in the file you include, you could even include binary files. This is not recommended... let's just say the behaviour of EBuild is undefined in that case.

Since every file is included before processing the build file constants cannot be used in the file name. The following scheme may illustrate it:

1. Parse the build file for any include directives and include the files.

2. Process the build file line by line and substitute constants in rules.

3. Build target if necessary and substitute constants in actions.

1.10 Ebuild.guide/Misc

```
Misc
****

   Once you get to know build, you'll discover you can use it for more
purposes than just this. See it as an intelligent script tool.

   If you want to find out the details of what build can do, read the
documentation of some unix-make, as build should be somewhat compatible
with this. What it doesn't do for now, is:

   - allow backslash at the end of a line for longer rules

   When EBuild discovers a cyclic dependancy it just aborts, i.e. this
won't be executed:

     bla: defs.m blurp.m bla
      ec $(target).e

   since the target bla.e has the dependancy bla.e. EBuild used to
crash with an infinite loop on this one. However, it is still very easy
to make an infinite loop:

     bla: defs.m blurp.m bla
      ec $(target).e

     defs.m: bla
             ec defs.e

   bla depends on defs.m which depends on bla which depends on defs.m
which depends on bla which depends on defs.m which depends on bla which
depends on defs.m ...
```

1.11 Ebuild.guide/Bugs - Future

```
Bugs / Future
*************

   Bugs: none known

   Future: implement some more 'Make' features. Top of things to do:

   * Allow for recursion in symbolic constants.

   * Implement some more preset symbols:
        * the first file this target depends on (done)

        * all but the first file this target depends on
```

```
        * ...

   *     #include-like directive (done)

   * yet another argument to list the targets available in the build
     file. (minor)
```

1.12 Ebuild.guide/History

```
History
*******

   For v3.1 it was updated by Jason Hulance, to fix the bug that
executed actions in reverse order. Also he introduced the local variable
$target in actions.

   EBuild was updated for v3.3a by Gregor Goldbach to support symbolic
constants and to stop on cyclic dependancies. The $target behaviour was
expanded to match other symbols: $(target) is legal, too.
```

1.13 Ebuild.guide/The Authors

```
The Authors
***********

   Wouter van Oortmerssen is the creator of E. He has studied computer
sciences and lives in England where he occasionally destroys monitors.

   Jason R Hulance is an Englishman and they say he has met Wouter
several times. He coded some tools for E, most notably Explorer which
runs together with EDBG in the current E release.

   Rob is just Rob.

   Gregor Goldbach loves E, started studying computer sciences in
October '97 and lives in Germany. He met Wouter but his monitor is
still running.

   The current maintainer of EBuild is Gregor Goldbach. Bug reports and
suggestions should be sent to him via email
(<glauschwuffel@amt.comlink.de>), you can also find him on the E
mailing list.

   This EBuild documentation is part of the Amiga E Encyclopedia which
can be found at
http://www.asta.uni-hamburg.de/users/goldi/aee/aee_1.html (online)
or on aminet/dev/e (snapshot).
```