NAME

acd − a compiler driver

SYNOPSIS

acd **−v**[*n*] **−vn**[*n*] **−name** *name* **−descr** *descr* **−T** *dir* [*arg* ...]

DESCRIPTION

Acd is a compiler driver, a program that calls the several passes that are needed to compile a source file. It keeps track of all the temporary files used between the passes. It also defines the interface of the compiler, the options the user gets to see.

This text only describes **acd** itself, it says nothing about the different options the C-compiler accepts. (It has nothing to do with any language, other than being a tool to give a compiler a user interface.)

OPTIONS

Acd itself takes five options:

−v[*n*]
Sets the diagnostic level to *n* (by default **2**). The higher *n* is, the more output **acd** generates: **−v0** does not produce any output. **−v1** prints the basenames of the programs called. **−v2** prints names and arguments of the programs called. **−v3** shows the commands executed from the description file too. **−v4** shows the program read from the description file too. Levels 3 and 4 use backspace overstrikes that look good when viewing the output with a smart pager.

−vn[*n*]
Like **−v** except that no command is executed. The driver is just play-acting.

−name *name*
Acd is normally linked to the name the compiler is to be called with by the user. The basename of this, say **cc**, is the call name of the driver. It plays a role in selecting the proper description file. With the **−name** option one can change this. **Acd −name cc** has the same effect as calling the program as **cc**.

−descr *descr*
Allows one to choose the pass description file of the driver. By default *descr* is the same as *name*, the call name of the program. If *descr* doesn't start with **/**, **./**, or **../** then the file **/usr/lib/***descr***/descr** will be used for the description, otherwise *descr* itself. Thus **cc −descr newcc** calls the C-compiler with a different description file without changing the call name. Finally, if *descr* is **"−"**, standard input is read. (The default lib directory **/usr/lib**, may be changed to *dir* at compile time by **−DLIB=\"***dir***\"**. The default *descr* may be set with **−DDESCR=\"***descr***\"** for simple installations on a system without symlinks.)

−T *dir*
Temporary files are made in **/tmp** by default, which may be overridden by the environment variable **TMPDIR**, which may be overridden by the **−T** option.

THE DESCRIPTION FILE

The description file is a program interpreted by the driver. It has variables, lists of files, argument parsing commands, and rules for transforming input files.

Syntax

There are four simple objects:

Words, Substitutions, Letters, and Operators.

And there are two ways to group objects:

Lists, forming sequences of anything but letters,

Strings, forming sequences of anything but Words and Operators.

Each object has the following syntax:

Words
They are sequences of characters, like **cc**, **−I/usr/include**, **/lib/cpp**. No whitespace and no special characters. The backslash character (\) may be used to make special characters common, except whitespace. A backslash followed by whitespace is completely removed from the input. The sequence **\n** is changed to a newline.

Substitutions
: A substitution (henceforth called 'subst') is formed with a **$**, e.g. **$opt**, **$PATH**, **${lib}**, **$∗**. The variable name after the **$** is made of letters, digits and underscores, or any sequence of characters between parentheses or braces, or a single other character. A subst indicates that the value of the named variable must be substituted in the list or string when fully evaluated.

Letters
: Letters are the single characters that would make up a word.

Operators
: The characters =, +, −, ∗, <, and > are the operators. The first four must be surrounded by whitespace if they are to be seen as special (they are often used in arguments). The last two are always special.

Lists
: One line of objects in the description file forms a list. Put parentheses around it and you have a sublist. The values of variables are lists.

Strings
: Anything that is not yet a word is a string. All it needs is that the substs in it are evaluated, e.g. **$LIBPATH/lib$key.a**. A single subst doesn't make a string, it expands to a list. You need at least one letter or other subst next to it. Strings (and words) may also be formed by enclosing them in double quotes. Only \ and **$** keep their special meaning within quotes.

Evaluation

One thing has to be carefully understood: Substitutions are delayed until the last possible moment, and description files make heavy use of this. Only if a subst is tainted, either because its variable is declared local, or because a subst in its variable's value is tainted, is it immediately substituted. So if a list is assigned to a variable then this list is only checked for tainted substs. Those substs are replaced by the value of their variable. This is called partial evaluation.

Full evaluation expands all substs, the list is flattened, i.e. all parentheses are removed from sublists.

Implosive evaluation is the last that has to be done to a list before it can be used as a command to execute. The substs within a string have been evaluated to lists after full expansion, but a string must be turned into a single word, not a list. To make this happen, a string is first exploded to all possible combinations of words choosing one member of the lists within the string. These words are tried one by one to see if they exist as a file. The first one that exists is taken, if none exists than the first choice is used. As an example, assume **LIBPATH** equals **(/lib /usr/lib)**, **key** is **(c)** and **key** happens to be local. Then we have:

"$LIBPATH/lib$key.a"

before evaluation,

"$LIBPATH/lib(c).a"

after partial evaluation,

"(/lib/libc.a /usr/lib/libc.a)"

after full evaluation, and finally

/usr/lib/libc.a

after implosion, if the file exists.

Operators

The operators modify the way evaluation is done and perform a special function on a list:

∗
: Forces full evaluation on all the list elements following it. Use it to force substitution of the current value of a variable. This is the only operator that forces immediate evaluation.

+
: When a + exists in a list that is fully evaluated, then all the elements before the + are imploded and all elements after the + are imploded and added to the list if they are not already in the list. So this operator can be used either for set addition, or to force implosive expansion within a sublist.

−
: Like +, except that elements after the − are removed from the list.

The set operators can be used to gather options that exclude each other or for their side effect of implosive expansion. You may want to write:

cpp −I$LIBPATH/include

to call cpp with an extra include directory, but **$LIBPATH** is expanded using a filename starting with **−I** so this won't work. Given that any problem in Computer Science can be solved with an extra level of indirection, use this instead:

cpp −I$INCLUDE
INCLUDE = $LIBPATH/include +

Special Variables

There are three special variables used in a description file: **$***, **$<**, and **$>**. These variables are always local and mostly read-only. They will be explained later.

A Program

The lists in a description file form a program that is executed from the first to the last list. The first word in a list may be recognized as a builtin command (only if the first list element is indeed simply a word.) If it is not a builtin command then the list is imploded and used as a UNIX command with arguments.

Indentation (by tabs or spaces) is not just makeup for a program, but are used to group lines together. Some builtin commands need a body. These bodies are simply lines at a deeper indentation.

Empty lines are not ignored either, they have the same indentation level as the line before it. Comments (starting with a **#** and ending at end of line) have an indentation of their own and can be used as null commands.

Acd will complain about unexpected indentation shifts and empty bodies. Commands can share the same body by placing them at the same indentation level before the indented body. They are then "guards" to the same body, and are tried one by one until one succeeds, after which the body is executed.

Semicolons may be used to separate commands instead of newlines. The commands are then all at the indentation level of the first.

Execution phases

The driver runs in three phases: Initialization, Argument scanning, and Compilation. Not all commands work in all phases. This is further explained below.

The Commands

The commands accept arguments that are usually generic expressions that implode to a word or a list of words. When *var* is specified, then a single word or subst needs to be given, so an assignment can be either *name* **=** *value*, or **$***name* **=** *value*.

var = *expr* ...
: The partially evaluated list of expressions is assigned to *var*. During the evaluation is *var* marked as local, and after the assignment set from undefined to defined.

unset *var*
: *Var* is set to null and is marked as undefined.

import *var*
: If *var* is defined in the environment of **acd** then it is assigned to *var*. The environment variable is split into words at whitespace and colons. Empty space between two colons (**::**) is changed to a dot.

mktemp *var* **[***suffix***]**
: Assigns to *var* the name of a new temporary file, usually something like /tmp/acd12345x. If *suffix* is present then it will be added to the temporary file's name. (Use it because some programs require it, or just because it looks good.) **Acd** remembers this file, and will delete it as soon as you stop referencing it.

temporary *word*
Mark the file named by *word* as a temporary file. You have to make sure that the name is stored in some list in imploded form, and not just temporarily created when *word* is evaluated, because then it will be immediately removed and forgotten.

stop *suffix*
Sets the target suffix for the compilation phase. Something like **stop .o** means that the source files must be compiled to object files. At least one **stop** command must be executed before the compilation phase begins. It may not be changed during the compilation phase. (Note: There is no restriction on *suffix*, it need not start with a dot.)

treat *file suffix*
Marks the file as having the given suffix for the compile phase. Useful for sending a **−l** option directly to the loader by treating it as having the **.a** suffix.

numeric *arg*
Checks if *arg* is a number. If not then **acd** will exit with a nice error message.

error *expr ...*
Makes the driver print the error message *expr* ... and exit.

if *expr* = *expr*
If tests if the two expressions are equal using set comparison, i.e. each expression should contain all the words in the other expression. If the test succeeds then the if-body is executed.

ifdef *var*
Executes the ifdef-body if *var* is defined.

ifndef *var*
Executes the ifndef-body if *var* is undefined.

iftemp *arg*
Executes the iftemp-body if *arg* is a temporary file. Use it when a command has the same file as input and output and you don't want to clobber the source file:

```
transform .o .o
	iftemp $*
		$> = $*
	else
		cp $* $>
	optimize $>
```

ifhash *arg*
Executes the ifhash-body if *arg* is an existing file with a '**#**' as the very first character. This usually indicates that the file must be pre-processed:

```
transform .s .o
	ifhash $*
		mktemp ASM .s
		$CPP $* > $ASM
	else
		ASM = $*
	$AS −o $> $ASM
	unset ASM
```

else
Executes the else-body if the last executed **if**, **ifdef**, **ifndef**, **iftemp**, or **ifhash** was unsuccessful. Note that **else** need not immediately follow an if, but you are advised not to make use of this. It is a "feature" that may not last.

apply *suffix1 suffix2*
Executed inside a transform rule body to transform the input file according to another transform rule that has the given input and output suffixes. The file under **$*** will be replaced by the new file. So if there is a **.c .i** preprocessor rule then the example of **ifhash** can be

replaced by:

```
transform .s .o
        ifhash $*
                apply .c .i
        $AS −o $> $*
```

include *descr*

Reads another description file and replaces the **include** with it. Execution continues with the first list in the new program. The search for *descr* is the same as used for the **−descr** option. Use **include** to switch in different front ends or back ends, or to call a shared description file with a different initialization. Note that *descr* is only evaluated the first time the **include** is called. After that the **include** has been replaced with the included program, so changing its argument won't get you a different file.

arg *string ...*

Arg may be executed in the initialization and scanning phase to post an argument scanning rule, that's all the command itself does. Like an **if** that fails it allows more guards to share the same body.

transform *suffix1 suffix2*

Transform, like **arg**, only posts a rule to transform a file with the suffix *suffix1* into a file with the suffix *suffix2*.

prefer *suffix1 suffix2*

Tells that the transformation rule from *suffix1* to *suffix2* is to be preferred when looking for a transformation path to the stop suffix. Normally the shortest route to the stop suffix is used. **Prefer** is ignored on a **combine**, because the special nature of combines does not allow ambiguity.

The two suffixes on a **transform** or **prefer** may be the same, giving a rule that is only executed when preferred.

combine *suffix-list suffix*

Combine is like **transform** except that it allows a list of input suffixes to match several types of input files that must be combined into one.

scan

The scanning phase may be run early from the initialization phase with the **scan** command. Use it if you need to make choices based on the arguments before posting the transformation rules. After running this, **scan** and **arg** become no-ops.

compile

Move on to the compilation phase early, so that you have a chance to run a few extra commands before exiting. This command implies a **scan**.

Any other command is seen as a UNIX command. This is where the **<** and **>** operators come into play. They redirect standard input and standard output to the file mentioned after them, just like the shell. **Acd** will stop with an error if the command is not successful.

The Initialization Phase

The driver starts by executing the program once from top to bottom to initialize variables and post argument scanning and transformation rules.

The Scanning Phase

In this phase the driver makes a pass over the command line arguments to process options. Each **arg** rule is tried one by one in the order they were posted against the front of the argument list. If a match is made then the matched arguments are removed from the argument list and the arg-body is executed. If no match can be made then the first argument is moved to the list of files waiting to be transformed and the scan is restarted.

The match is done as follows: Each of the strings after **arg** must match one argument at the front of the argument list. A character in a string must match a character in an argument word, a subst in a string may match 1 to all remaining characters in the argument, preferring the shortest possible match. The hyphen in a argument starting with a hyphen cannot be matched by a subst. Therefore:

arg −i

matches only the argument **−i**.

arg −O$n

matches any argument that starts with **−O** and is at least three characters long. Lastly,

arg −o $out

matches **−o** and the argument following it, unless that argument starts with a hyphen.

The variable **$∗** is set to all the matched arguments before the arg-body is executed. All the substs in the arg strings are set to the characters they match. The variable **$>** is set to null. All the values of the variables are saved and the variables marked local. All variables except **$>** are marked read-only. After the arg-body is executed is the value of **$>** concatenated to the file list. This allows one to stuff new files into the transformation phase. These added names are not evaluated until the start of the next phase.

The Compilation Phase

The files gathered in the file list in the scanning phase are now transformed one by one using the transformation rules. The shortest, or preferred route is computed for each file all the way to the stop suffix. Each file is transformed until it lands at the stop suffix, or at a combine rule. After a while all files are either fully transformed or at a combine rule.

The driver chooses a combine rule that is not on a path from another combine rule and executes it. The file that results is then transformed until it again lands at a combine rule or the stop suffix. This continues until all files are at the stop suffix and the program exits.

The paths through transform rules may be ambiguous and have cycles, they will be resolved. But paths through combines must be unambiguous, because of the many paths from the different files that meet there. A description file will usually have only one combine rule for the loader. However if you do have a combine conflict then put a no-op transform rule in front of one to resolve the problem.

If a file matches a long and a short suffix then the long suffix is preferred. By putting a null input suffix ("") in a rule one can match any file that no other rule matches. You can send unknown files to the loader this way.

The variable **$∗** is set to the file to be transformed or the files to be combined before the transform or combine-body is executed. **$>** is set to the output file name, it may again be modified. **$<** is set to the original name of the first file of **$∗** with the leading directories and the suffix removed. **$∗** will be made up of temporary files after the first rule. **$>** will be another temporary file or the name of the target file (**$<** plus the stop suffix), if the stop suffix is reached.

$> is passed to the next rule; it is imploded and checked to be a single word. This driver does not store intermediate object files in the current directory like most other compilers, but keeps them in **/tmp** too. (Who knows if the current directory can have files created in?) As an example, here is how you can express the "normal" method:

```
transform .s .o
        if $> = $<.o
                # Stop suffix is .o
        else
                $> = $<.o
                temporary $>
        $AS −o $> $∗
```

Note that **temporary** is not called if the target is already the object file, or you would lose the intended result! **$>** is known to be a word, because **$<** is local. (Any string whose substs are all expanded changes to a word.)

Predefined Variables

The driver has three variables predefined: **PROGRAM**, set to the call name of the driver, **VERSION**, the driver's version number, and **ARCH**, set to the name of the default output architecture. The latter is optional, and only defined if **acd** was compiled with **−DARCH=\"***arch-name***\"**.

EXAMPLE

As an example a description file for a C compiler is given. It has a front end (ccom), an intermediate code optimizer (opt), a code generator (cg), an assembler (as), and a loader (ld). The compiler can pre-process, but there is also a separate cpp. If the **−D** and options like it are changed to look like **−o** then this example is even as required by POSIX.

```
# The compiler support search path.
C =     /lib /usr/lib /usr/local/lib

# Compiler passes.
CPP =   $C/cpp $CPP_F
CCOM =          $C/ccom $CPP_F
OPT =   $C/opt
CG =    $C/cg
AS =    $C/as
LD =    $C/ld

# Predefined symbols.
CPP_F =         −D__EXAMPLE_CC__

# Library path.
LIBPATH = $USERLIBPATH $C

# Default transformation target.
stop .out

# Preprocessor directives.
arg −D$name
arg −U$name
arg −I$dir
        CPP_F = $CPP_F $*

# Stop suffix.
arg −c
        stop .o

arg −E
        stop .E

# Optimization.
arg −O
        prefer .m .m
        OPT = $OPT -O1

arg −O$n
        numeric $n
        prefer .m .m
        OPT = $OPT $*

# Add debug info to the executable.
arg −g
        CCOM = $CCOM -g
```

```
# Add directories to the library path.
arg −L$dir
        USERLIBPATH = $USERLIBPATH $dir

# −llib must be searched in $LIBPATH later.
arg −l$lib
        $> = $LIBPATH/lib$lib.a

# Change output file.
arg −o$out
arg −o $out
        OUT = $out

# Complain about a missing argument.
arg −o
        error "argument expected after ’$∗’"

# Any other option (like −s) are for the loader.
arg −$any
        LD = $LD $∗

# Preprocess C-source.
transform .c .i
        $CPP $∗ > $>

# Preprocess C-source and send it to standard output or $OUT.
transform .c .E
        ifndef OUT
                $CPP $∗
        else
                $CPP $∗ > $OUT

# Compile C-source to intermediate code.
transform .c .m
transform .i .m
        $CCOM $∗ $>

# Intermediate code optimizer.
transform .m .m
        $OPT $∗ > $>

# Intermediate to assembly.
transform .m .s
        $CG $∗ > $>

# Assembler to object code.
transform .s .o
        if $> = $<.o
                ifdef OUT
                        $> = $OUT
        $AS −o $> $∗

# Combine object files and libraries to an executable.
combine (.o .a) .out
        ifndef OUT
```

```
OUT = a.out
$LD −o $OUT $C/crtso.o $∗ $C/libc.a
```

FILES

/usr/lib/*descr*/descr − compiler driver description file.

SEE ALSO

cc(1).

ACKNOWLEDGEMENTS

Even though the end result doesn't look much like it, many ideas were nevertheless derived from the ACK compiler driver by Ed Keizer.

BUGS

POSIX requires that if compiling one source file to an object file fails then the compiler should continue with the next source file. There is no way **acd** can do this, it always stops after error. It doesn't even know what an object file is! (The requirement is stupid anyhow.)

If you don't think that tabs are 8 spaces wide, then don't mix them with spaces for indentation.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)