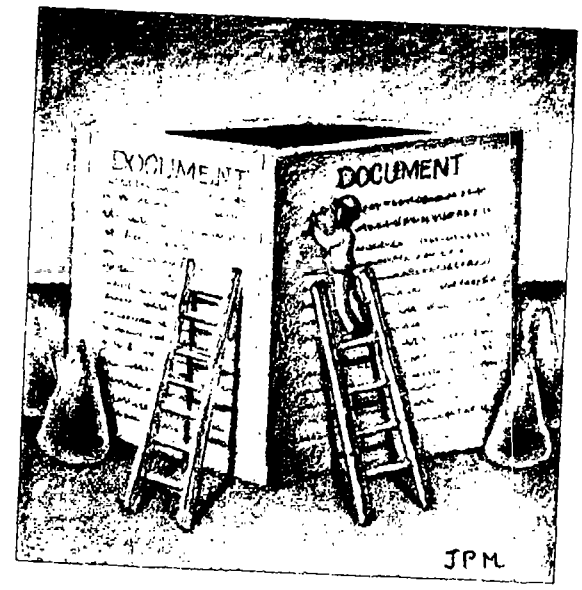
Building Stuff

by Jeffreys Copeland and Haemer

Right now tulips are blooming while the last snow melts, students at the university are shedding their down vests in favor of shorts, the Boulder Creek bicycle path is once again impassable at lunch: Spring has arrived in the Rockies. And with the arrival of spring, a young consultant's fancy turns to...a new way to build programs.

We promised two months ago to finish our exploration of how to build documents that can be formatted by both Web browsers and traditional paper formatters. In the verbal tradition of Alan Winston, systems manager at Stanford University's Synchrotron Radiation Laboratory, we've gotten distracted and said "push!" in the middle of a paragraph.

In this case, the thing in our new stack space was the `atchange` utility we wrote for Tom Schneider, a research biologist at the National Cancer Institute. Before we say "pop," we wanted to pass on some comments from Tom, now that he's been using `atchange` for a while, and to finish some thoughts that `atchange` raised around the water cooler. Besides, we've interrupted our flow of thought in every series we've written for *RS/Magazine*, so now it's become a tradition.



Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

Fan Mail from Some Flounder

We had a nice note from Tom after we sent him last month's article, commenting on our text. Tom rejected our suggestion that he should switch from the C shell to the Korn shell because he's already lost enough research time between the holidays and his visit to Boulder.

Tom also wondered why we were twitting him about continuing to program in Pascal: After all, he'd rather keep up with his research than rewrite an existing 200,000-line code base. (Copeland sympathizes: He's got a vital utility program that's been in use constantly since 1986, but he doesn't have a week to convert it from Pascal to C++, which is what it should be written in.) Tom also suggested that we add more comments in our `Perl` programs to help folks like him who are still learning the language. We'll try.

Tom pointed out an interesting feature of `atchange`. If you set `atchange` to watch a *directory*, you can trigger an action when any file in the directory changes. For example, if you're working on multiple utilities with multiple interacting sources in a single directory, `atchange` can fire off the `make` command to rebuild all the pieces in the directory.

Children of make–the UNIX Standby

Speaking of `make`, the old UNIX standby has been a mainstay of development work for nearly two decades. It's a useful tool. Before its invention by Stu Feldman, users had to keep track of what needed to be recompiled every time they edited a source file. Our pre-`make` work was to write scripts that just rebuilt everything, so we wouldn't screw up. It was not only a waste of programmer time, but also profligate of CPU power. Remember, this was back in the days of 64-KB main memory, when the bytes were still sometimes *core* memory.

Push! Copeland had a bottle of real memory cores on his desk at Interactive Systems Corp. for a number of years. Finally, one day someone asked what they were, which led to the exercise of walking around the office asking 50 or so technical people–folks who'd been programming for most of their professional lives, if not most of their chronological lives–what these little metal donuts were. Two people knew: Ted Dolotta and Marv Rubenstein, both of whom had written code for machines with *tubes*, a distinction neither of us can claim. Pop!

Anyway, `make` changed the way we built programs. Now, every directory of source has a `makefile` in it. In fact, because the `makefile` is not very well suited to interactions with source management tools, the System V.3 source tree, as delivered by AT&T, contains a small directory for each utility–`awk`, `cat`, `ls` and so on–most of which contain a `makefile` and a single source file.

A Directory of Little Utilities

We've noted the problem ourselves. We have a directory of little utilities, which in the normal course of events contains a `makefile` and an `RCS` directory. We check out a utility to work on and end up typing `make foo` a lot because the default target in the `makefile` is to rebuild everything. Worse still, we *can't* rebuild everything because most of the sources are safely locked up in their RCS archives. (Note that the Free Software Foundation's `gnumake` is better about RCS files.)

Well, in one of those cool ideas in the "now that you've told me, it's bloody obvious" category, back in

Figure 1

```
Makefile          RCS/              SHAR/             jfold.c           volume.c

./RCS:
Makefile,v        date-diff.c,v     pr.c,v            trunc.c,v
a2lf,v*           duplex,v*         recol,v*          uncram,v*
a2ps,v*           grepmail,v*       psfixtoc,v*       unvt100.c,v-DEAD
backup,v*         headers.c,v       purge,v*          unvt100.l,v
backup-full,v*    jfold,v*          soelim.c,v        uucat.c,v
booksort,v*       jfold.c,v         sound,v*          uudecode.c,v
cram,v*           pagecnt.ps,v      tex,v*            uuencode.c,v
daily,v*          pclbreak.c,v      texit,v*          volume.c,v
daily-disk,v*     pclmend.c,v       tps,v*            wdiff,v*

./SHAR:
date-parse.jlc    date-parse.sh     rel.shar
```

1989 David MacKenzie at Rockefeller University in New York published a program in comp.sources.misc (Volume 6, Issue 9, at ftp://ftp.uu.net/usenet/comp.sources.misc/volume6/xc.Z) that solved the problem. His program, the idea for which he says he stole from a similar program at St. Olaf College in Northfield, MN, assumes that the command to build a simple, one-module utility is buried in header comments of the program source itself. For example, an ls -CF -R of our small sources directory looks like Figure 1.

Imagine how much easier it would be to have our programs begin like the example below, rather than have to rely on a 300-line makefile.

```
static char id[] = "$Id: rdr.c,v 1.7 96/01/30
23:17:42 jeff Exp $";
/* CMD: $(CC) $(CFLAGS) -o rdr rdr.c */
```

Notice that we're using something that looks a lot like the syntax from our makefile.

How do we process this line? We'll break tradition and show a shell script rather than our usual Perl program in the next section.

The Build Program

We'll run through the actual code for the program, one paragraph at a time. We will begin with the normal header declarations, including the magic cookie to declare this a shell script.

```
#! /bin/sh
# Basic build script
# $Id: bld.new,v 1.3 1996/02/05 04:03:36 jeff Exp $
```

Next, we've saved the last source file name in $HOME/.bldlog, so we can use it as the default source file on this invocation. This means that after we've started, instead of typing make and relying on the makefile to contain the appropriate default rules, we type bld and rely on our log of recent sources for a default.

We could just save the name of the last source file, but we're smarter than that. We tag each saved file name with the directory, so we can work on several files in parallel, each in a different directory. This is probably overkill, but it's the strategy we use for our front-end shell function for vi: In that case, we really do want to have a separate target for each directory.

```
##    first, find the current default
##       file for this directory:
where=`pwd`
[ -r $HOME/.bldlog ] || touch $HOME/.bldlog
file=`sed -n "s;^$where ;;p" $HOME/.bldlog`
#      use the file named on the command line,
#         if we have one...
[ -r "$1" ] && file=$1 && shift
```

We also want to derive the target name from the source name. Between source and target, we now have the contents of the make variables $< and $@, which we can go ahead and use in our command lines.

```
#   we set the default target, too
target=`expr $file : '\( .*\)\.[^.]*' `
```

We could have screwed up and not put a file name on the command line:

```
##  check for error
[ -z "$file" ] && echo no file specified?! && exit
[ ! -r "$file" ] &&  \
    echo "can't read source file $file" && exit
```

Now that we know what the source file is, we need to extract the build rule from that file:

```
##  now extract the command line
##     from the source file
sed -n "s;/\**CMD: *\ (.*\) \*/;\1;p" $file >/tmp/$$
```

And similarly, if there isn't a command line, we create a default line.

```
#   we may need the default command line,
#   if what we initially extracted was null:
[ ! -s /tmp/$$ ] &&
   echo '${CC} ${CFLAGS} -o $@ $< ${LIBS}' >/tmp/$$
```

Now comes a slightly tricky bit. We're allowing ourselves to use environment variables in the command line. This is very important: As with a regular makefile, we want to allow ourselves the ability to change the default compiler by changing a definition. This is particularly important when moving sources from machine to machine.

We run the Sun SPARCworks compiler on our SunOS boxes but the Free Software Foundation GNU compiler on our Solaris machines. We do this by having three sets of definitions, which are just shell environment variable sets.

The first set of definitions is hard-wiring for the default values of compiler and flags. The second is a global per-user file of definitions. Both of these can be overridden by the third: definitions in the local directory.

Why several levels? Suppose, for example, that we have some software in a pc subdirectory that we compile

with a DOS cross-compiler, or we want to set LIBS=-lm in our math utility subdirectory.

We are, in effect, creating a shell script to run our compiler, so these definition files are pretty straightforward. Our $HOME/.bldrc consists of just shell environment sets:

```
CFLAGS=-g
CC=gcc
LIBS=
```

We just cat to grab those files that exist. (Note that we are grouping all these commands together in parentheses so that their output is sent together down an output pipe.)

```
##  we have defaults and then
##    global and local setup files
( echo CC=cc CFLAGS=-O LIBS=
    [ -r $HOME/.bldrc ] && cat $HOME/.bldrc
    [ -r ./.bldrc ] && cat ./bldrc
```

Note that we've opened a block with a parenthesis above, that we'll close way below. As a check, we turn the echo for the command line itself. This allows us to see the compile line that's beginning to run, just like our old, familiar make.

```
#       turn on expanded echo
    echo set -x
```

We have our build command squirreled away in /tmp/$$, and we should be able to add it to the stream for our output pipe directly. No such luck. We've got two variables that we can't use shell environment variables for, $@ and $<, so we do the substitution by directly editing the build command.

```
##    now make substitutions for the $@ and $<,
##    which we can't get from the shell
    sed -e "s/\$@/$target/g" \
        -e "s/\$</$file/g" /tmp/$$) |
sh
```

What happens to the output we've pushed down this pipe? It goes to a simple destination: the Bourne shell, sh. We're almost done. We need to save the name of the source file that we've just built in our log and do a little cleanup.

```
##    log the file for later use
( echo $where $file;
  egrep -v "^$where " $HOME/.bldlog) >/tmp/$$
mv /tmp/$$ $HOME/.bldlog
```

Voilá: We're done. Next month, we plan to get back to HTML. Until then, happy trails. ▲

READER FEEDBACK

To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service card located elsewhere in this magazine. Rate the following column and feature topics in this issue.

	INTEREST LEVEL		
Features:	High	Medium	Low
Managing in the Distributed World	170	171	172
Reviews: Network Backup Software and a Tape Library	173	174	175
Columns:			
Q&AIX–What's New with GNU?	176	177	178
Systems Wrangler–Watchdogs for the Herd	179	180	181
Datagrams–rdist: An Old and Reliable Tool	182	183	184
AIXtensions–Data Breathing Room–AIX 4.1 File Systems	185	186	187
Work–Building Stuff	188	189	190